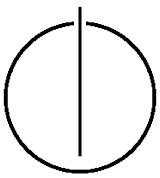
FAKULTÄT FÜR INFORMATIK

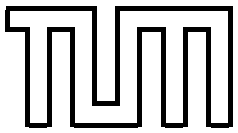
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Cross-layer Data-centric Usage Control

Enrico Lovat





FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl XXII - Software Engineering

Cross-layer Data-centric Usage Control

Enrico Lovat

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Helmut Seidl
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Alexander Pretschner
2.	Prof. Dr. Frank Piessens, Katholieke Universiteit Leuven, Belgium

Die Dissertation wurde am 13.05.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.06.2015 angenommen.

Acknowledgments

First and foremost, I shall thank my supervisor, Prof. Dr. Alexander Pretschner. Without your guidance, your example and our loud discussions, probably no part of this work would have ever been completed. Your sharpness and conciseness, especially in scientific writing, the ethic and the passion you put in teaching, the intellectual honesty of admitting when you are wrong and the strength of your arguments when you are not, are only some of the aspects of your leadership that I admire and plan to emulate in the future.

My second thanks goes to my second supervisor, Prof. Dr. Frank Piessens. Thanks for providing me extremely valuable feedback; I am grateful for the opportunity of benefiting from your clever advices and my main regret is not having exploited it more often.

Thanks to the German Research Foundation (DFG), which supported this work through the Priority Program 1496 "Reliably Secure Software Systems RS^3", grants PR-1266/1-2-3.

I would like to thank some of my fellow colleagues and my co-authors for their direct contributions to this work. In particular, thank you Florian, Johan, Martin, Prachi and Tobias for positively engaging in those stimulating never-ending discussions; together with conferences and paper reviews, those represent my idea of scientific research.

Thanks to Alexander, Christoph, Cornelius and Sebastian for their picky reviews on the first drafts of this dissertation; you definitely improved its quality.

Thanks to all the Bachelor and Master students who did their thesis under my supervision and whose results, mentioned in this work, confirmed the feasibility of the theoretical models I have been working on.

I would like to express my gratitude also to some individuals who contributed, at various stages of my life, to my choice of becoming a computer scientist: thanks Santa Claus, for bringing me my first PC when I was nine; thanks Nico, for making me fall in love with programming before I was legally allowed to drive a scooter; thanks A., for laughing at me in front of my parents, during a vocational guidance event, when I said that I would like to attend a technical high school and become a computer scientist; thanks Prof. Minio, for teaching me how to think instead of how to write code; and thank you K. for making sure that a Master degree was not enough.

Without these people's intervention, I would probably be a carpenter, a bartender or an underpaid code monkey. Most likely a bartender.

Last but not least, I want to thank my family for the unconditional support I received over these years: Thanks to my parents, who made me the man I am today with their education and their hard-working example, I did not forget that. Thanks to my wife, for giving up her life to blindly follow me in this adventure; I appreciate all the sacrifices you did to let me pursue this goal and, without your support, this work would have never been possible. And finally, thanks to my beautiful daughter: You are the reason why I arrive late in the morning and I leave early in the evening, taking precious time from my work. You are also the reason why the moment I arrive home marks only half of my working day. And yet, you are the reason why nevertheless I am looking forward to come home every night. Thank you for rearranging my priorities.

Abstract

Usage control is concerned with what happens to data after access has been granted. In the literature, usage control models have been defined on the grounds of *events* that, somehow, are related to data. Within the system, data (e.g. "a picture") may exist in form of different representations, possibly located at different layers of abstraction (e.g. a file, a set of pixels in a window, a Java object, a memory region, etc...). In order to protect data, one must observe and control the usage of all its different representations.

This research proposes a usage control model and a language that capture the distinction between *data* and *representations* of data. In this framework, policies can constraint the usage of single representations (e.g., delete file1.jpg after thirty days) as well as all representations of the same data (e.g., if file2.jpg is a copy of file1.jpg, also delete file2.jpg), even at different layers of abstraction (e.g. if a process is rendering the content of file1.jpg in a window, close the window and kill the process).

The core contribution of this work is the formalization and implementation of the first generic model that integrates usage control and cross-layer data flow tracking, and it is presented in the first part of this work.

The second part addresses the problem of refining the precision of the framework by augmenting the data flow tracking component with additional information about the system. In particular, this dissertation analyzes three approaches that leverage, respectively, information about structure of data, amount of data and results from static information flow analysis. All the solution are formalized, implemented and evaluated, and represent the second major contribution of this work.

Zusammenfassung

Nutzungskontrolle beschäftigt sich mit der Frage, was mit Daten nach deren initialer Freigabe passieren darf und soll. In der Literatur wurden Modelle der Nutzungskontrolle bisher stets auf Basis von *Ereignissen* definiert, die einen mehr oder weniger klaren Bezug zu den zu kontrollierenden Daten haben. Daten treten innerhalb eines Systems allerdings in verschiedenen Repräsentationen, und potentiell auf verschiedenen Abstraktionsebenen (z.B. als Datei, als Menge von Pixeln in einem Fenster, als Java Objekt, oder als Region im Hauptspeicher), auf. Um die Nutzung eines Datums kontrollieren zu können, müssen daher alle Repräsentationen des Datums überwacht und kontrolliert werden.

In der vorliegenden Arbeit wird ein Modell und eine Spezifikationssprache für ein Konzept der Nutzungskontrolle vorgeschlagen, das explizit zwischen *Daten* und deren *Repräsentationen* unterscheidet. Mit diesem Konzept ist es möglich, sowohl Policies bezüglich einzelner Repräsentationen (z.B. "lösche file1.jpg innerhalb von 30 Tagen"), als auch aller Repräsentationen eines Datums (z.B. "falls file2.jpg eine Kopie von file1.jpg ist, lösche auch file2.jpg") zu spezifizieren und diese auf verschiedenen Abstraktionsebenen durchzusetzen (z.B. "falls ein Prozess den Inhalt von file1.jpg in einem Fenster anzeigt, dann schließe dieses Fenster und beende den entsprechenden Prozess").

Der erste Teil der Arbeit beschäftigt sich mit dem Hauptbeitrag der selbigen, nämlich der Formalisierung und Implementierung des ersten generischen Modells, das abstraktionsübergreifende Datenflussverfolgung in das Konzept der Nutzungskontrolle integriert.

Der zweite Teil der Arbeit adressiert das Problem der Verfeinerung der Präzision durch Adaption der Datenflussverfolgungskomponenten mit zusätzlichen Informationen über das betrachtete System. Insbesondere diskutiert diese Arbeit drei Ansätze, die die Präzision des Konzepts durch Analyse der Struktur, Berücksichtigung der Quantität von Daten, sowie durch die Integration von Ergebnissen aus statischer Informationsflussanalyse verbessern. Die Formalisierung, Implementierung, und Evaluation dieser Ansätze stellt den zweiten Hauptbeitrag dieser Arbeit dar.

Contents

Outline of the Thesis

CHAPTER 1: INTRODUCTION

This chapter is an introduction to the topic and to the fundamental issues addressed by this thesis. It discusses motivation, context, assumptions and limitations of the solutions presented in this work.

Part I - Representation-Independent Data Usage Control

The first part describes the major contribution of this thesis: a formal model and a language for data usage control that addresses representations of data at and across different layers of abstraction.

CHAPTER 2: BACKGROUND: USAGE CONTROL

This chapter describes a formal model and a language for usage control, mostly based on results from the literature [76], that represents the starting point for the work described in these pages. Part of the content is taken from an unpublished work co-authored by the author of this dissertation [137].

CHAPTER 3: USAGE CONTROL MARRIES DATA FLOW TRACKING

This chapter describes a generic model and language to augment the usage control concepts of Chapter 2 with data flow tracking features. This work is one of the core contributions of the thesis and has been published in [138]. The discussion on soundness is taken from [109], an unpublished work co-authored by the author of this dissertation.

CHAPTER 4: CROSS-LAYER DATA FLOW TRACKING

This chapter describes a generic model to track flows of data across different instances of the model described in Chapter 3. This work represents the second major contribution of the thesis and is part of two unpublished works co-authored by the author of this dissertation [109, 137].

CHAPTER 5: SYSTEM DESIGN AND IMPLEMENTATION

This chapter describes a generic architecture to instantiate the concepts described in the previous chapters. Despite some similarities with the XACML [143] system model, this architecture is part of the author's contribution and has been published in [138] and [100], both co-authored by the author of this dissertation.

Part II - Taming Label Creep: Enhanced Data Flow Tracking

The second part of the work describes three different models for data flow tracking that enhance the precision of the model described in Part I. Each solution is implemented and evaluated in terms of precision and performance, also with respect to the basic model.

CHAPTER 6: STRUCTURED DATA FLOW TRACKING

This chapter describes an extension of the model in which the tracking precision is improved using information about the structure of data. A preliminary version of this work

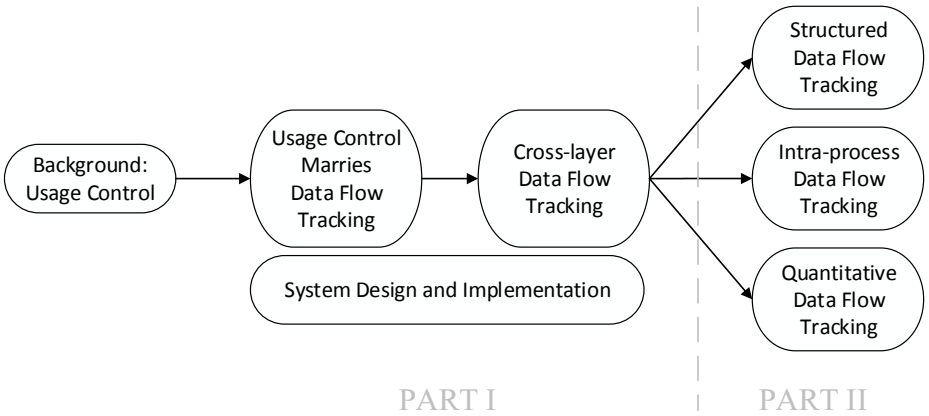
has been published in [108], a publication co-authored by the author of this dissertation. The implementation and evaluation of the model are part of this thesis's original contribution.

CHAPTER 7: INTRA-PROCESS DATA FLOW TRACKING

This chapter presents a second solution to improve the precision of the model introduced in Chapter 3. In this work, co-authored by the author of this dissertation and published in [107], the goal is achieved by leveraging static information flow analysis results.

CHAPTER 8: QUANTITATIVE DATA FLOW TRACKING

This chapter presents a third extension of Chapter 3's model. In this work, co-authored by the author of this dissertation and published in [110], the basic model is augmented with information about the amount of data transferred by system events. This supports more expressive policies and containers declassifcation criteria to mitigate overapproximation issues.



CHAPTER 9: RELATED WORK

This chapter discusses related work from the literature, including author's prior work.

CHAPTER 10: CONCLUSIONS

This chapter summarizes the conclusions of this thesis and presents possible future work that could stem from the presented results.

N.B.: *Almost every chapter of this dissertation is based on different publications authored or co-authored by the author of this dissertation. Such publications are mentioned in the short descriptions above and described in more details in Section 9.1. Due to the obvious content overlapping, quotes from such publications within the respective chapters are not marked explicitly.*

1. Introduction

From the picture uploaded on a social network, to the access log of an enterprise server, from intelligence information on a secret services' laptop to the home address associated to a supermarket's loyalty card, the amount and variety of sensitive data stored and processed by today's computing systems has reached levels that were unimaginable only two decades ago. The emergence of mobile and ubiquitous computing, together with the rise of cloud-based services and technologies, contributed to drastically increase the amount of personal and other generic sensitive data produced. The central role of information technologies in this scenario emphasizes the importance of solutions to control access to and usage of sensitive data.

Historically, the discipline studying how to express and enforce selective restrictions of access to certain resources (i.e. data) was *access control* [148]. However, the purpose of access control ends once a party gets access to the data. The extension of access control that also investigates the problem of specifying and enforcing what may, must and must not happen to data *after* access to it has been granted is called *Usage Control*.

Examples of usage control requirements are "delete this file after 30 days", "don't copy this picture" or "notify owner of this data upon access". Given its overwhelming presence in today's multimedia, perhaps the most common and well-known instance of usage control is Digital Right Management (DRM) technologies [12, 116, 133].

Usage control is relevant in different domains, including the management of intellectual property, privacy, the management of secrets, and compliance with regulations. In terms of privacy, users may want, among other things, to control exactly how their medical information, loyalty card records, telecommunication connection records, and banking information are used and by whom. This has become more and more an issue with the growth of popularity of largely unprotected social network sites, where users intentionally or inadvertently publish huge amount of personal data. At least in Europe, in addition to EU-wide legal requirements, current developments clearly indicate a potential or actual political fallout when large sets of citizen or customer data become available [135].

In terms of intellectual property, the increasing trend of implementing business processes in a distributed manner via outsourcing requires the exchange of confidential information between companies, like blueprints, results of experiments, etc.. Companies are obviously interested in such data being used solely according to their expectations, especially when such companies are administrations, state departments, intelligence agencies or the military, who want to have control over how secret information is disseminated.

Finally, the need for usage control is also reflected in regulations. In Europe, an example is the EU directive 95/46/EC [3], which requires data to be collected and used only according to a specified purpose; in the United States, the Sarbanes-Oxley Act (SOX) [2] requires specific data to be retained for five years; and the US Health Insurance Portability and Accountability Act (HIPAA) [1] stipulates strict documentation requirements on the dissemination of data [106].

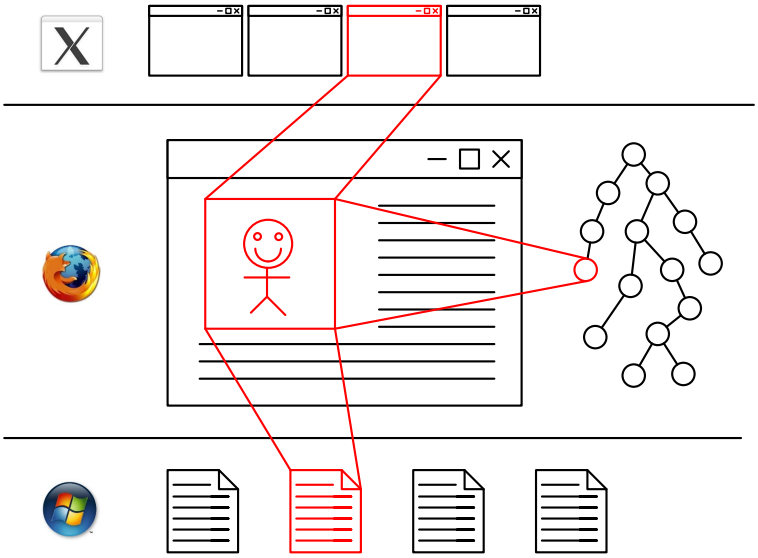


Figure 1.1.: Example of data (social network profile picture) that exists in form of multiple representations at different layers of abstraction: as a set of pixel at the windowing level (top), as a DOM element within the memory of the browser (center), and as a cache file at the operating system layer (bottom).

Usage control and enforcement mechanisms have historically been defined on the ground of *events*, leveraging techniques such as complex event processing [112] and runtime verification [102]. Policy languages for expressing usage control requirements have been investigated in the last decade by several authors [47, 11, 18, 128, 173, 76, 40, 154, 160] and tailored usage control solutions have been proposed for specific layers of abstraction, including machine languages [52, 166], Java [39, 86], enterprise service bus [63, 126], dedicated applications such as Internet Explorer [49] and Mozilla Firefox [100].

The goal of the work presented in this dissertation is to investigate a different aspect of the problem of policy specification and enforcement: the *data* dimension. While requirements are usually expressed in terms of specific representations, like "`do not print pic.jpg`", it is often the case that the policy writer's actual intention is to protect *the abstract data*, e.g. "the picture", and not just one of its *representations*, e.g. a file. The fundamental difference is that while in both cases file `pic.jpg` could not be printed, a literal interpretation of the policy implies that a copy of the file or a screenshot of the window in which the file is rendered could.

To enforce such a requirement, one must consider that a picture may exist in multiple representations within the system, which potentially reside at different layers of abstraction, like network packets, window pixmaps, Java objects, data base records, or files, as depicted in the example in Figure 1.1.

Moreover, the system may contain multiple copies of the same file or multiple clones of an object. Usage control requirements usually concern *integrity* and *confidentiality* prop-

erties. While the former refer to one or some specific representations (e.g. "*content of file /etc/passwd cannot be modified*", but any copy of it can), the latter typically address all the representations of the same data at once (e.g. "*this report cannot leave the local system*", no matter which copy).

The general issue with existing usage-control solutions [129, 126, 76, 143, 62, 63, 40] is their focus on events, ignoring the fact that the same sensitive data may exist within the system in multiple representations. The fundamental problem addressed by this work, thus, is whether and how it is possible to build a usage control framework that allows for the specification of requirements that address all the different representations of the same data at once and that defines how such policies should be enforced.

1.1. Goal

The goal of this work is a usage control framework that is at least as expressive as existing solutions from the literature, allowing for the specification and enforcement of all the relevant kinds of requirements, such as:

- rights and duties ("*do not print*" and "*notify admin if copying*"),
- temporal constraints ("*delete after thirty days*"),
- event-defined conditions ("*report can be disseminated only after being officially approved by the CEO*"),
- cardinality constraints ("*at most five copies*"),
- spatial constraints ("*within Germany*"),
- purpose conditions ("*for statistical purposes only*") and
- environment conditions pertaining to organization ("*compliance with ISO 17799*") and technology ("*open only if personal firewall activated*").

Such requirements, usually based on law and regulations, self-regulation, and economic interest, have been proven to be relevant by different requirements elicitation studies [80, 77, 82, 78].

Moreover, the outcome framework must be aware of all the different representations of data within the system and supports specification and enforcement of policies on all of them at once ("*this picture can not be printed*", no matter which copy of it). In addition to be faithful to the real system, the framework should be as precise as possible in identifying the different representations of data without, at the same time, significantly impacting on the performances of the overall system, i.e. the system usability should not be compromised. Considering that the notion of "the system is still usable" depends on the specific instantiation contexts, the framework should also allow for different configurations of precision/performance tradeoff.

1.2. Research Description

1.2.1. Problem

To achieve the goal, the following research questions must be addressed:

- How can usage control requirements be specified and enforced over all the representations of the same data at once?

- How can the relation between data representations and events at different layers be modeled? What determines flows of data across different layers of abstraction?
- What does it mean that a certain data flow tracking solution is "sound"?
- How can the precision of the data flow tracking model be augmented with additional information about the system? In particular,
  - What role can the structure of data play in this task?
  - How can results from static information flow analysis help?
  - How can the amount of data in the different representations be quantified? How can it be used to improve precision?
  - How can the improvement in terms of precision be quantified?
- What is the overhead induced by the framework? Which aspects of the system affect it and how?

1.2.2. Thesis Statement

This work tries to answer the questions in Section 1.2, confirming the thesis that

> **It is possible to formalize and to realize a sound and non-trivial generic usage control framework to constraint the usage of data in form of all its different representations, at and across different layers of abstraction, with acceptable performance overhead.**

More precisely,

TS-1: It is possible to formalize a system that supports expression and enforcement of usage control requirements on different representations of the same data at once, even when data is disseminated across different layers of abstractions;

TS-2: Because of its generic nature, such model can be instantiated and implemented for different layers of abstraction;

TS-3: The model is sound, i.e. correct, and non-trivially precise. In other words, the model is more precise than trivial approaches like "every data is possibly stored in every container", which are sound but useless in practical terms;

TS-4: The precision of the enforcement can be adjusted by augmenting the model with additional information about the system;

TS-5: The performance overhead imposed by the implementations of the model is acceptable, i.e. depending on the use case, it does not compromise the overall system functionality.

1.2.3. Solution Strategy

The goal will be achieved in three major steps, that aim at answering all the research questions in Section 1.2:

1. Extending a generic usage control model from the literature [76, 81] with data flow tracking features, to relate the different representations of data within the system. The result will be a generic model instantiatable at arbitrary layers of abstraction in the system.[1]

2. Defining a generic model to track flows of data across different instances of the model defined in step 1, possibly across different layers of abstraction (*cross-layer data flow tracking*) in a non-trivial overapproximating way. This will support integration of enforcement mechanisms at different layers which would in turn provide system-wide usage control.

3. Refining the results of steps 1 and 2, in terms of precision e.g. considering data flow tracking approaches that accounts for structural or quantitative aspects of data or leveraging other forms of analysis, like static information flow analysis. Each different solution will be implemented and evaluated, both in terms of precision and performance.

1.2.4. Contribution

The contributions of this work are:

- The first generic language for policy specification that supports the distinction between *data* and *representations* of data (Chapter 3);
- The first generic model for data usage control, instantiatable at arbitrary layers of abstraction, that can enforce such policies (Chapter 3);
- The first generic model for system-wide data flow tracking, based on the combination of multiple instantiations of the model at different layers of abstraction (Chapter 4);
- A definition of soundness for data flow tracking for a generic layer of abstraction with respect to canonical notions from information theory (Section 3.4) ;
- A definition of soundness for data flow tracking across multiple layers of abstraction and a proof of soundness of the generic model for cross-layer data flow tracking (Section 4.3.4);
- Some well-defined notions of precision that allow for quantitative comparison of different data flow tracking models (Section 6.5.5, Section 7.3.2 and Section 8.5.2);
- Three possible extensions of the model for data flow tracking and their evaluations, that confirm the possibility of tracking precision improvement by introducing additional information about the system and with minimal performance overhead (Chapter 6, Chapter 7 and Chapter 8).

[1] The formal language used as basis for this work [76] already supports the first four types of requirements mentioned in Section 1.1. In terms of environment conditions and purpose and spatial constraints, respective information is assumed to exist in form of externally available attributes, encoded as propositions in the language (eval operator, see Chapter 2).

1.3. System Model and Assumptions

In this work, usage control is assumed to take place in the context of systems like those described in [81, 76]. In these models, a *data consumer*, $\mathfrak{c}$, requires access to a certain data d from a *data provider*, $\mathfrak{p}$. Together with its request, $\mathfrak{c}$ provides a description of its enforcement capabilities. After performing access control checks [103] to make sure $\mathfrak{c}$ is entitled to get access to d, $\mathfrak{p}$ checks whether $\mathfrak{c}$ is also able to enforce the usage control restrictions associated to d, specified in policy π. In case $\mathfrak{c}$ is not able to enforce π, different strategies can be applied, like ignoring $\mathfrak{c}$'s request, modifying π to a more relaxed version, sending a declassified (e.g. anonymized) version of d to $\mathfrak{c}$ or using other form of enforcement outside the digital world (e.g. non-disclosure agreements contracts). If both access and usage control checks succeed, instead, $\mathfrak{p}$ sends d to $\mathfrak{c}$, together with a policy π to be enforced by $\mathfrak{c}$. When $\mathfrak{c}$ receives d, its local usage control infrastructure takes care of deploying π and enforcing it on every usage of d.

The work described in this dissertation can be located at this point of the negotiation process: it focuses on the policies and the enforcement mechanisms, and it assumes all of the above negotiation and deployment steps to have already taken place. It also assumes that the execution of every event that affects or makes use of sensitive data is mediated by the usage control infrastructure. This happens by means of monitoring technologies based on two entities, *signalers* and *monitors*. Signalers, like the system call interposition framework described in Section 3.3.2, make events visible to the monitor, which consume them, check adherence to the deployed policies, and possibly take specified actions, like denying the execution of the event, modifying the value of some parameter or executing compensative actions. This process is described in detail in Chapter 5.

The possibility of modifying or denying the execution of a certain event requires the signaler to be capable of notifying the monitor about the *intention* of executing a certain event before the event is actually executed. This is a fundamental requirement in order to enforce usage control policies in a *preventive* fashion, i.e. to make sure that violations of the policy never take place. If the signaler is not capable of changing or denying the execution of a certain event or if it is only capable of observing events after they already happened, usage control can only be enforced in a *detective* fashion, and only compensative additional actions can be taken (e.g. "*notify the system administrator*").

Finally, if $\mathfrak{c}$ redistributes data d to a third party, the negotiation process is assumed to take place again, in this case with $\mathfrak{c}$ in the role of data producer. Note that $\mathfrak{c}$ can only distribute d with an associated policy that is at least as restrictive as π, i.e. with no less duties and no more permissions. The integration of existing work from the literature that tackles the problem of policy evolution [139, 134] with the model presented in the next chapters is a straightforward exercise, but it has been intentionally left out of this work for the sake of simplicity and clarity in the explanation.

In usage controlled systems, an attacker is an entity that attempts to use data in ways that differ from what is specified by the respective policy. Depending on the intention of the attacker, it is possible to distinguish two categories of policy violations: those performed by motivated attackers that *intentionally* try to circumvent restrictions and misuse the data, and those due to *unintentional* behaviors of the user of the system that accidentally tries to misuse data (e.g. trying to print a classified document on a public printer). While this distinction is irrelevant at the model level, in the latter case it is possible to make more

relaxed assumptions about the system. For instance, even if the user has enough permissions to kill the usage control process, it is reasonable to assume that this will not happen, because it is in the user's interest that the usage control framework works properly. Conversely, in order to cope with the first class of attackers, the remaining of this work assumes also the following to hold:

- all the implementations of the the monitors for the different layers of abstraction, the usage control infrastructure, the storage and the communication channels for the framework metadata (e.g. in-between different monitors) are correctly implemented and free of any vulnerability that might allow attackers to obtain illegitimate permissions or to perform side channels attacks, e.g. switching off the complete usage control infrastructure;
- the usage control solutions is always up and running and not tampered with, and
- any other running software on the system cannot interfere with the execution of the usage control solutions, e.g. by hiding relevant data usage events from the signaler.

1.4. Running Scenario

The following scenario is going to be used as a reference in the remaining of this work:

> *Alice works as analyst at ACME Inc., a smartphone manufacturer. All sensitive projects at ACME are stored in a central repository, accessible by each client machine, like Alice's. Often enough, Alice requires information about new models under development. Her job includes analyzing it and comparing it with data from field experiments and from various public sources to the end of writing reports for suppliers and for other departments. Such projects are ACME's main asset and the primary challenge for the company is to contain the leakage of sensitive information.*

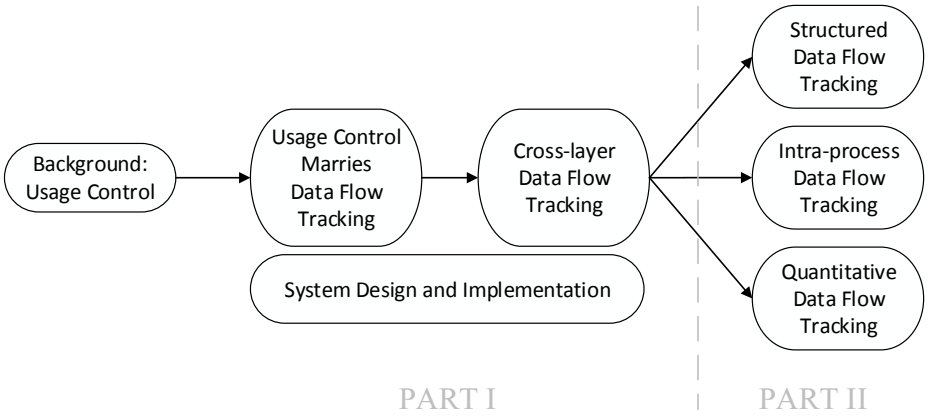
In the next chapters, different elaborations and specializations of this generic scenario will be used to illustrate how, under different circumstances, and at what price, the various approaches described in this work can help mitigating the data leakage issue.

1.5. Structure

The remaining of this work is divided in two parts.

The first part describes a model for event-driven usage control (Chapter 2) and shows how it can be extended with basic data flow tracking capabilities (Chapter 3). The model is then extended to allow multiple instantiations to cooperate (Chapter 4). The architecture of the framework and a discussion about concrete instantiations of the model is presented in Chapter 5.

The second part of this dissertation discusses three approaches to refine the precision of the data flow tracking component. Each of these solutions leverages additional information about the system, specifically: structure of data (Chapter 6), static information flow analysis results (Chapter 7) and amount of data transferred by system events (Chapter 8).



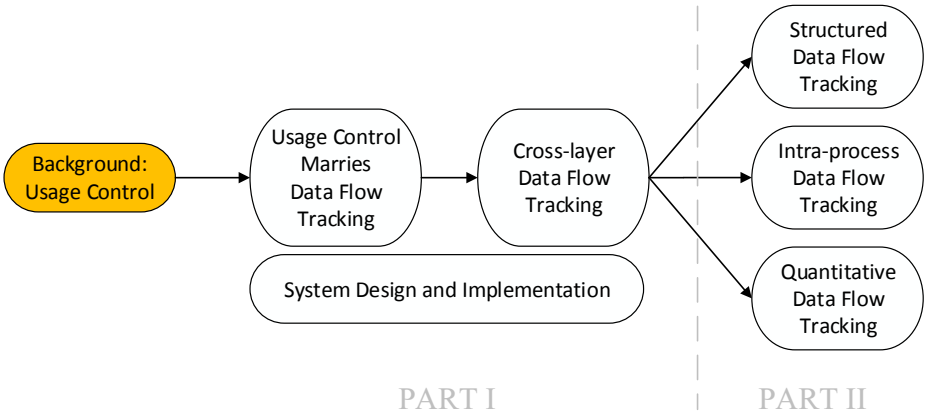
The evaluation of each solution and the tradeoff additional precision/performance overhead is discussed individually within the respective chapters.

In Chapter 9 the relevance of the work presented in this dissertation is justified by comparison with related work from the literature. Previous work from the author is also presented in the same chapter (Section 9.1). Finally, limitations and strategies to possibly overcome them in future work are discussed in Chapter 10. A concrete syntax for the data usage control language can be found in in Appendix A, together with the detailed proofs of soundness of the cross-layer model and of the simplification rules of the quantitative data flow tracking model (Chapter 8), respectively in Appendix B and Appendix C.

Part I
Representation-Independent Data Usage Control

2. Background: Usage Control

This chapter describes a formal model and a language for usage control, mostly based on results from the literature [76], that represents the starting point for the work described in these pages. Part of the content is taken from an unpublished work co-authored by the author of this dissertation [137].



Usage Control policies specify constraints on traces of events, defined in Section 2.1. The language used to specify them is roughly a linear temporal logic with some macros for cardinality operators. The second part of this chapter presents and discusses the differences between specification-level policies (see Section 2.2), which describe *what* the constraints are, and implementation-level policies (see Section 2.3), which specify *how* the usage control requirements should be enforced. Finally, Section 2.4 discusses how to activate policy enforcement and how to react to policy violation and Section 2.5 concludes with an overview of different aspects related to detective enforcement.

2.1. System Model

2.1.1. Events

Events (set $\mathcal{E}$) are defined by a name (set $EName$) and a set of parameters. Parameters are defined by a name (set $PName$) and a value (set $PValue$). Formally,

$$\mathcal{E} \subseteq EName \times \mathbb{P}(PName \times PValue).$$

For an event $e \in \mathcal{E}$, let $e.n$ denote the event's name and $e.p$ the set of its (parameter, value) pairs.

For reasons that will become more clear in Chapter 3, $e.p$ always contains a parameter $obj \in PName$ that denotes the primary target object of event e. For example,

$$print \mapsto \{obj \mapsto report.xls, quality \mapsto 100, pages \mapsto \text{1-5}\} \in \mathcal{E}$$

specifies that the first five pages of $report.xls$ should be printed at 100% quality. In policies, events may occur with variables that map names (set $VName$) to possible values (set $VVal$).

$$Var = VName \to VVal \qquad \text{for } VVal \subseteq \mathbb{P}(PValue \cup EName).$$

Given a set of variables, the set of possibly variable events $\mathcal{VE}$ is then defined as

$$\mathcal{VE} \subseteq (EName \cup VName) \times \mathbb{P}(PName \times (PValue \cup VName))$$

where variables are assumed to respect the types, i.e., to range over event names or parameter values. For instance, for $Var = \{v_1 \mapsto \{report1.xls, report2.xls\}, v_2 \mapsto \{51, \ldots, 100\}\}$, the variable event $print \mapsto \{obj \mapsto v_1, quality \mapsto v_2\} \in \mathcal{VE}$ represents any event that prints either $report1.xls$ or $report2.xls$ with a quality greater than 50%. In the following, $VarsIn(t)$ will denote the set of variables in a term t.

2.1.2. Refinement

When one wants to specify a constraint on an event, some parameters may be more relevant than others. For instance, it may be relevant that *report1.xls* has been printed, but the quality with which it has been printed may be irrelevant. This is captured by the refinement relation between non-variable events, which states that an event e_1 refines an event e_2 if e_1 and e_2 have the same name and if the value of every parameter specified in e_2 has the same value in e_1, i.e. if the set of pairs (parameter name, parameter value) provided by e_2 is a subset of that for e_1. Formally,

$$\begin{aligned} &refinesEv \subseteq \mathcal{E} \times \mathcal{E} \\ &\forall e_1, e_2 \in \mathcal{E} : e_1 \; refinesEv \; e_2 \Leftrightarrow e_1.n = e_2.n \wedge e_1.p \supseteq e_2.p \end{aligned}$$

For instance, given three events,

$$\begin{aligned} e_a &= print \mapsto \{obj \mapsto report.xls, quality \mapsto 80\} \\ e_b &= print \mapsto \{obj \mapsto report.xls\} \\ e_c &= print \mapsto \{obj \mapsto report.xls, quality \mapsto 100\}), \end{aligned}$$

according to the definition, e_a is a refinement of e_b ($e_a \; refinesEv \; e_b$), but e_a is not a refinement of e_c, because of the disagreement on the quality parameter. Event refinement is very useful in policy specification because it allows to only specify values for relevant parameters, assuming a universal quantification over those that are not mentioned.

In contrast, when a system is running and an event actually "happens", all its parameters are specified, i.e. they have a value. The event is then called a "maximum refinement". The set of such maximally refined events, $maxRefEv$, is defined as

$$maxRefEv = \{e \in \mathcal{E} \mid \nexists e' \in \mathcal{E} : \; e' \neq e \wedge e' \; refinesEv \; e\}.$$

2.1.3. Semantic Model of Events and System Runs

Enforcing policies in a preventive fashion (see Section 1.3) requires the capability of distinguishing between the *intention* of executing an event and its *actual* concrete execution. For

instance, the event triggered by clicking a button in a graphical user interface is different from the execution of command associated to the button; the first event can be considered the "intention" of executing the second. Note that if one can only observe events that already happened, one would need to be able to undo events in order to perform preventive enforcement, and that is often impossible.

Since in a system run all parameters are specified when an event takes place, the set of system events is defined as

$$\mathcal{S} \subseteq maxRefEv \times \{intended, actual\}.$$

System runs are modeled as traces,

$$Trace : \mathbb{N} \to \mathbb{P}(\mathcal{S}),$$

that map abstract moments in time to sets of system events.

Note that for simplicity's sake, events that are observed within the same timestep are assumed to be *independent* to each other, meaning that the state at the beginning of the following timestep is the same regardless of the order in which they are executed. While this may sound like a restrictive assumption, in concrete experience with actual implementations (see Section 3.3) it has always been possible to establish a total order between events within the same timestep by leveraging additional information, e.g. system timestamp. For this reason, the instantiations described in Section 3.3 are defined in terms of single events rather than sets.

2.2. Specification-Level Policies

Specification-level policies express constraints on the system events in form of trace properties, in contrast to implementation-level policies (see Section 2.3), which describe how specification-level policies should be enforced. Specification-level policies are described by first order temporal formulae.

2.2.1. Propositions

The first order propositional part of policies is defined by

$$\begin{aligned} \Gamma ::= &\quad \mathcal{VE} \mid \mathbb{N} \mid String \mid \Gamma\ op\ \Gamma \mid \ldots \\ \Psi ::= &\quad (\Psi) \mid \underline{false} \mid \Psi\ \underline{implies}\ \Psi \mid E(\mathcal{VE}) \mid I(\mathcal{VE}) \mid eval(\Gamma) \mid \underline{forall}\ VName\ \underline{in}\ VVal : \Psi. \end{aligned}$$

For shortness of notation, let common shortcuts in Ψ be defined as follows:

- $\underline{true}$ for $\underline{false}\ \underline{implies}\ \underline{false}$,
- $\underline{not}(\psi)$ for $\psi\ \underline{implies}\ \underline{false}$,
- $\psi_1\ \underline{or}\ \psi_2$ for $(\underline{not}\ \psi_1)\ \underline{implies}\ \psi_2$, and
- $\psi_1\ \underline{and}\ \psi_2$ for $\underline{not}(\psi_1\ \underline{implies}\ \underline{not}\ \psi_2)$, and
- $\underline{exists}\ vn\ \underline{in}\ VS : \psi$ for $\underline{not}(\underline{forall}\ vn\ \in\ VS : \underline{not}\ \psi)$.

Whenever no confusion between semantics and syntax is possible, in the following the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used for *implies*, *not*, *and*, *or*, *forall*, *exists*, *in*.

$I(\cdot)$ and $E(\cdot)$ syntactically capture the distinction between intended and actual events introduced in the previous subsection.

In order to keep formalization more clean and concise, the additional distinction between first and ongoing events, treated by previous work [76], is not considered here. This distinction was used to disambiguate the semantics of terms like "*play the song at most twice*", which could be interpreted as (1) "*play the song twice*" (no matter for how long) as well as (2) "*play the song for at most two moments in time*" (no matter if consecutively or in two different sessions). However, this distinction does not introduce any new fundamental challenge, and, as such, it has been explicitly ignored in this work.

2.2.2. Conditions Outside Temporal and First Order Logic

$eval$ is used for the specification of conditions on actual events. A full language Γ for the respective computation is not provided here. One choice could be XPath, that was also used in the implementations described in [105, 138]. As an example, with $vn_1, vn_2 \in VName$,

$$\begin{aligned} &Var = \{vn_1 \mapsto \{0, \ldots, 100\}, vn_2 \mapsto \{simple, matte, glossy, photo\}\} \\ &E(print \mapsto \{obj \mapsto catalog.xls, quality \mapsto vn_1, paper \mapsto vn_2\}) \\ &\qquad\qquad \rightarrow \big(eval(vn_2 \neq photo) \ \rightarrow \ eval(vn_1 \leq 70)\big) \end{aligned}$$

specifies that $catalog.xls$ cannot be printed on non-photographic paper with quality higher than 70%. The use of $eval$ makes it possible to specify conditions on (absolute) time and location, one of the requirements described in Section 1.1. This has, for instance, been implemented [55] for mobile phones to the end of enforcing requirements such as "do not use outside company premises." The semantics of $eval(\gamma)$ is left unspecified and referred to as

$$[\![eval(\gamma)]\!]_{eval}.$$

2.2.3. Formal Semantics of Events

The semantics of events is defined by $\models_e \subseteq \mathcal{S} \times \Psi$ as follows: For a variable event e that contains a variable $v \in Var$ with $vn = dom(v)$ and for $x \in Var(vn)$, let $e[vn \mapsto x]$ denote the result of replacing all occurrences of the variable's name vn in e by the value x. More than one substitution may be specified within the square brackets. Then, let

$$Inst_\varepsilon : \mathcal{VE} \rightarrow \mathbb{P}(\mathcal{E})$$

define a function that generates all ground substitutions of an event with variables, i.e.,

$$\begin{aligned} &\forall e \in \mathcal{VE}, vn_1 \ldots vn_k \in VName, VS_1 \ldots VS_k \in VVal : \\ &\quad VarsIn(e) = \{vn_1 \mapsto VS_1, \ldots, vn_k \mapsto VS_k\} \\ &\qquad\qquad \Rightarrow Inst_\varepsilon(e) = \{e[vn_1 \mapsto vv_1, \ldots, vn_k \mapsto vv_k] \mid \textstyle\bigwedge_{i=1}^{k} vv_i \in VS_i\}. \end{aligned}$$

The semantics of possibly variable events is defined as

$$\begin{aligned}&\forall e' \in maxRefEv; e \in \mathcal{VE}\\ &\qquad (e', actual) \models_e E(e) \iff \exists e'' \in \mathcal{E} : e'\ refinesEv\ e'' \wedge e'' \in Inst_\varepsilon(e)\\ \wedge\quad &(e', intended) \models_e I(e) \iff \exists e'' \in \mathcal{E} : e'\ refinesEv\ e'' \wedge e'' \in Inst_\varepsilon(e).\end{aligned}$$

2.2.4. First Order Future Time Formulae

Overloading the propositional operators, the abstract syntax of the future temporal logic for specifying usage control requirements is defined as

$$\begin{aligned}\Phi ::= \quad &(\Phi) \mid \Psi \mid \Phi\ \underline{implies}\ \Phi \mid \underline{forall}\ VName\ \underline{in}\ VVal : \Phi \mid\\ &\Phi\ \underline{until}\ \Phi \mid \Phi\ \underline{after}\ \mathbb{N} \mid \underline{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{repuntil}(\mathbb{N}, \Psi, \Phi)\end{aligned}$$

plus the macros $\underline{true}, \underline{not}(\varphi), \varphi_1\ \underline{and}\ \varphi_2, \varphi_1\ \underline{or}\ \varphi_2, \underline{exists}\ vn\ \underline{in}\ VV : \varphi$ as described in Section 2.2.1. Whenever no confusion between semantics and syntax is possible, in the following the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used for $\underline{implies}$, $\underline{not}$, $\underline{and}$, $\underline{or}$, $\underline{forall}$, $\underline{exists}$, $\underline{in}$. The intuitive semantics of the propositional and first order operators is as usual:

- $\underline{until}$ is the weak until operator of LTL: $\varphi_1\ \ \underline{until}\ \varphi_2$ is true iff φ_1 is true until φ_2 eventually becomes true, or φ_1 is true eternally;
- $\varphi\ \underline{after}\ n$ is true iff φ becomes true after n timesteps;
- $\underline{replim}(n, l, r, \psi)$ is true if ψ is true in-between l and r times within the next n steps;
- $\underline{repuntil}(n, \psi, \chi)$ is true iff ψ is true at most n times until χ eventually becomes true, or until eternity if χ never becomes true.

It is convenient to add four further shortcuts to the syntax:

- $\Box\varphi$ for $\varphi\ \underline{until}\ \underline{false}$ specifies that φ will always be true in the future;
- $n\ \underline{repmax}\ \psi$, defined as $\underline{repuntil}(n, \psi, \underline{false})$, specifies that ψ should be true at most n times in the future;
- $\psi\ \underline{within}\ n$ for $\underline{replim}(n, 1, n, \psi)$ and $\psi\ \underline{during}\ n$ for $\underline{replim}(n, n, n, \psi)$ for $n \in \mathbb{N}, \varphi \in \Phi, \psi \in \Psi$ describe that ψ will become true at least once ($\underline{within}$) or continuously ($\underline{during}$) in the next n steps.

Lifting the notation for substitutions of events, for a formula $\varphi \in \Phi$ that contains a variable $vn \in Var$, the formula $\varphi[vn \mapsto vv]$ will denote the result of substituting all occurrences of vn in φ by value $vv \in Var(vn)$.

The formal semantics is defined by $\models_f \subseteq (Trace \times \mathbb{N}) \times \Phi$ as follows.

$$
\begin{aligned}
&\forall t \in Trace; n \in \mathbb{N}; \varphi \in \Phi : (t,n) \models_f \varphi \Leftrightarrow \varphi \neq \underline{false} \wedge \\
&\quad \big(\exists e \in \mathcal{VE} : (\varphi = E(e) \vee \varphi = I(e)) \wedge \exists e' \in t(n) : e' \models_e \varphi \\
&\quad \vee \exists \psi, \chi \in \Phi : \varphi = \psi \ \underline{implies} \ \chi \wedge \neg((t,n) \models_f \psi) \vee (t,n) \models_f \chi \\
&\quad \vee \exists \gamma \in \Gamma : \varphi = eval(\gamma) \wedge [\![\varphi]\!]_{eval} = true \\
&\quad \vee \exists vn \in VName; vs \in VVal; \psi \in \Phi \ : \\
&\qquad \varphi = (\underline{forall} \ vn \ \underline{in} \ vs : \psi) \wedge \forall vv \in vs : (t,n) \models_f \psi[vn \mapsto vv] \\
&\quad \vee \exists \psi, \chi \in \Phi : \varphi = \psi \ \underline{until} \ \chi \wedge \big(\forall v \in \mathbb{N} : n \leq v \Rightarrow (t,v) \models_f \psi \\
&\qquad \vee \exists u \in \mathbb{N} : n < u \wedge (t,u) \models_f \chi \wedge \forall v \in \mathbb{N} : n \leq v < u \Rightarrow (t,v) \models_f \psi\big) \\
&\quad \vee \exists m \in \mathbb{N}; \psi \in \Phi : \varphi = \psi \ \underline{after} \ m \wedge (t, n+m) \models_f \psi \\
&\quad \vee \exists m \in \mathbb{N}_1; l, r \in \mathbb{N}; \psi \in \Psi : \varphi = \underline{replim}(m,l,r,\psi) \wedge \\
&\qquad l \leq \#\{j \in \mathbb{N}_1 \mid j \leq m \wedge (t, n+j) \models_f \psi\} \leq r \\
&\quad \vee \exists m \in \mathbb{N}; \psi \in \Psi, \chi \in \Phi : \varphi = \underline{repuntil}(m, \psi, \chi) \\
&\qquad \wedge \big((\exists u \in \mathbb{N}_1 : (t, n+u) \models_f \chi \wedge (\forall v \in \mathbb{N}_1 : v < u \Rightarrow \neg((t, n+v) \models_f \chi)) \\
&\qquad\quad \wedge (\#\{j \in \mathbb{N}_1 \mid j \leq u \wedge t(n+j) \models_f \psi\}) \leq m) \\
&\qquad\quad \vee (\#\{j \in \mathbb{N}_1 \mid t(n+j) \models_f \psi\}) \leq m\big)\big)
\end{aligned}
$$

2.3. Implementation-Level Policies

A specification-level policy describes a particular usage control requirement that has to be satisfied, i.e. *what* should be enforced. With an implementation-level policies (ILP), one specifies *how* this should happen. In general, this can be achieved by inhibition, modification, and execution [132]: consider for instance the policy "*non-anonymized data cannot leave the system without notifying the administrator*". Adherence to this policy can be ensured at the implementation level in different ways, like by denying any attempt of sending non-anonymized data outside the system (inhibition) or by notifying the administrator (execution of an external event) or actually anonymizing the data when data is about to leave the system (modification).

ILPs for preventive enforcement (detective enforcement is discussed in Section 2.5) can be described as event-condition-action rules [132]: if the *intention* of executing a certain event e, denoted as $I(e)$, is detected and allowing its execution would make the condition φ true, then one of the following compensative actions is performed:

Inhibition The intended event e is not converted into an actual event ($\neg E(e)$).

Execution Event e may be executed, unless other ILPs prohibit it, and a further set of events, $\bigwedge_{i=1}^{n} I(x_i)$, is executed in addition. Since these events themselves can be subject to treatment by ILPs, they are specified as intended and not actual events.

Modification The execution of e is allowed, but in a modified form, e.g. changing the value of a certain parameter. This is equivalent to inhibiting the execution of e and executing its modified version e'. In general, a modifier is modeled as the combination of an inhibitor (for e) and an executor (of e' or, in general, of an arbitrary set of events).

2.3.1. Past Time Conditions

At runtime, decisions about a certain event can be taken only based on the trace of events that already happened, i.e. on the *past events*. For this reason, conditions for ILPs are specified in a past variant of language Φ, called Φ^-, defined as

$$\begin{aligned}\Phi^- ::= \quad & (\Phi^-) \mid \Psi \mid \Phi^- \ \underline{implies}^- \ \Phi^- \mid \underline{forall}\ VName\ \underline{in}\ VVal : \Phi^- \mid \\ & \Phi^- \ \underline{since}^- \ \Phi^- \mid \Phi^- \ \underline{before}^- \ \mathbb{N} \mid \underline{replim}^-(\mathbb{N},\mathbb{N},\mathbb{N},\Psi) \mid \underline{repsince}^-(\mathbb{N},\Psi,\Phi^-)\end{aligned}$$

plus the shortcuts $\underline{true}^-$, $\underline{not}^-$, $\underline{and}^-$, $\underline{or}^-$, $\underline{exists}$. Once again, if no confusion between semantics and syntax is possible, in the following the symbols $\rightarrow, \neg, \wedge, \vee, \forall, \exists, \in$ will be used for $\underline{implies}^-$, $\underline{not}^-$, $\underline{and}^-$, $\underline{or}^-$, $\underline{forall}$, $\underline{exists}$, $\underline{in}$. As in the future case, it is convenient to add four further shortcuts to the syntax:

- $\boxdot\varphi$ for $\varphi \ \underline{since}^- \ \underline{false}$, which is true if φ is true in all timesteps before and including the current timestep;

and, for $n \in \mathbb{N}, \varphi \in \Phi$ and $\psi \in \Psi$,

- $\psi \ \underline{repmax}^- \ n$ for $\underline{repsince}^-(n, \psi, \underline{false})$, to stipulate that ψ must have happened at most n times in the past;
- $\psi \ \underline{within}^- \ n$ for $\underline{replim}^-(n, 1, n, \psi)$, to stipulate that ψ has happened at least once in the past n steps;
- $\psi \ \underline{during}^- \ n$ for $\underline{replim}^-(n, n, n, \psi)$, which stipulates that ψ must have been true in each of the past n steps.

Lifting the notation for substitutions of events, for a formula $\varphi \in \Phi^-$ that contains a variable $vn \in Var$, let $\varphi[vn \mapsto vv]$ denote the result of substituting all occurrences of vn in φ by value $vv \in Var(vn)$. The semantics of Φ^- is defined by the infix operator $\models_{f^-} \subseteq (Trace \times \mathbb{N}) \times \Phi^-$ defined as:

$$\begin{aligned}
&\forall t \in Trace; n \in \mathbb{N}; \pi \in \Phi^- : (t,n) \models_{f^-} \pi \Leftrightarrow (\pi \neq \underline{false}) \wedge \\
&\quad \big(\exists e \in \mathcal{VE} : (\pi = E(e) \vee \pi = I(e)) \wedge \exists e' \in t(n) : e' \models_e \pi \\
&\vee \quad \exists \psi, \chi \in \Phi^- : \pi = \psi \ \underline{implies}^- \ \chi \wedge \neg((t,n) \models_{f^-} \psi) \vee (t,n) \models_{f^-} \chi \\
&\vee \quad \exists \gamma \in \Gamma : \pi = eval(\gamma) \wedge [\![\pi]\!]_{eval} = true \\
&\vee \quad \exists vn \in VName; vs \in VVal; \psi \in \Phi^- : \\
&\qquad \pi = (\underline{forall}\ vn\ \underline{in}\ vs : \psi) \wedge \forall vv \in vs : (t,n) \models_{f^-} \psi[vn \mapsto vv] \\
&\vee \quad \exists \psi, \chi \in \Phi^- : \pi = \psi \ \underline{since}^- \ \chi \wedge \big((\forall v \in \mathbb{N} : v \leq n \Rightarrow (t,v) \models_{f^-} \psi) \\
&\qquad \vee (\exists u \in \mathbb{N} : u \leq n \wedge (t,u) \models_{f^-} \chi \wedge \forall v \in \mathbb{N} : u < v \leq n \Rightarrow (t,v) \models_{f^-} \psi)\big) \\
&\vee \quad \exists m \in \mathbb{N}; \psi \in \Phi^- : \pi = \psi \ \underline{before}^- \ m \wedge n \geq m \wedge (t, n-m) \models_{f^-} \psi \\
&\vee \quad \exists m, l, r \in \mathbb{N}; \psi \in \Psi; : \varphi = \underline{replim}^-(m,l,r,\psi) \\
&\qquad \wedge\, l \leq (\#\{j \in \mathbb{N} \mid j \leq min(m,n) \wedge t(n-j) \models_{f^-} \psi\}) \leq r \\
&\vee \quad \exists m \in \mathbb{N}; \psi \in \Psi; \chi \in \Phi; e \in \mathcal{E} : \varphi = \underline{repsince}^-(m, \psi, \chi) \\
&\quad \wedge \big((\exists u \in \mathbb{N}_1 : n \geq u \wedge (t, n-u) \models_{f^-} \chi \wedge (\forall v \in \mathbb{N} : v < u \Rightarrow \neg((t, n-v) \models_{f^-} \chi)) \\
&\qquad \wedge (\#\{j \in \mathbb{N} \mid j \leq u \wedge t(n-j) \models_{f^-} \psi\} \leq m)) \\
&\quad \vee (\#\{j \in \mathbb{N} \mid j \leq n \wedge t(n-j) \models_{f^-} \psi\} \leq m)\big)\big)
\end{aligned}$$

2.3.2. ECA Rules

Let $tr \subseteq \mathcal{VE} \times \Phi^-$, defined as

$$tr(ve, \varphi) \Leftrightarrow \big(I(ve) \wedge (E(ve) \rightarrow \varphi)\big) \tag{2.1}$$

denote the situation (trigger event ve and condition $\varphi \in \Phi^-$) under which a specific ILP is supposed to fire. The model distinguishes between three different kinds of ILPs, based on the kind of actions they perform:

- $m_{inh} \subseteq \mathcal{VE} \times \Phi^-$ for the inhibitors,
- $m_{exc} \subseteq \mathcal{VE} \times \Phi^- \times \mathbb{P}(\mathcal{VE})$ for executors, and
- $m_{mod} \subseteq \mathcal{VE} \times \Phi^- \times \mathbb{P}(\mathcal{VE})$ for the modifiers.

Let $\forall\, VarsIn(\varphi)$ be a shortcut for $\forall vn_1 \in VV_1 : \ldots : \forall vn_k \in VV_k$ if $VarsIn(\varphi) = \{vn_1, \ldots, vn_k\}$ and $Var(vn_i) = VV_i$ for all $i \leq k$. ILPs are defined as

$$\begin{array}{lcll} m_{inh}(ve, \varphi) & \Leftrightarrow & \forall\, VarsIn(ve): & tr(ve, \varphi) \rightarrow \neg E(ve) \\ m_{exc}(ve, \varphi, Exc) & \Leftrightarrow & \forall\, VarsIn(ve): & tr(ve, \varphi) \rightarrow \bigwedge_{x_i \in Exc} I(x_i) \\ m_{mod}(ve, \varphi, Mod) & \Leftrightarrow & \forall\, VarsIn(ve): & tr(ve, \varphi) \rightarrow (\neg E(ve) \wedge m_{exc}(ve, \varphi, Mod)). \end{array}$$

Note that inhibitors can be expressed as modifiers via $m_{inh}(ve, \varphi) \leftrightarrow m_{mod}(ve, \varphi, \emptyset)$. The choice of using a slightly redundant notation here is motivated by readability's sake. For fixed sets of n_1 inhibiting, n_2 modifying and n_3 executing ILPs, their composition computes to

$$M \leftrightarrow \bigwedge_{i=1}^{n_1} m_{inh}(ve_i^{inh}, \varphi_i^{inh}) \wedge \bigwedge_{i=1}^{n_2} m_{mod}(ve_i^{mod}, \varphi_i^{mod}, Mod_i) \wedge \bigwedge_{i=1}^{n_3} m_{exc}(ve_i^{exc}, \varphi_i^{exc}, Exc_i). \tag{2.2}$$

2.3.3. Default Behavior

The final step is to allow intended events to become actual events, as long as no other ILP forbids them; in other words, the default policy is allow on all events. If there is an intended event $I(e)$, in a real system, it is always maximally refined, $e \in maxRefEv$. e should be allowed, i.e. $E(e)$ should be executed, if no modifying or inhibiting ILP fires. Such an ILP fires if e refines any ground substitution of its trigger event and, at the same time, makes the ILP's condition true. Taken together, the default rule is expressed as follows. If there is a maximally refined intended event, then either there must be a corresponding actual event; or a modifying or inhibiting ILP is triggered, which would prohibit the corresponding actual event. Formally,

$$M_{default} \leftrightarrow \bigwedge_{e \in maxRefEv} I(e) \rightarrow \Big(E(e) \vee \bigvee_{\substack{(ve, \varphi):\\ M \rightarrow m_{inh}(ve, \varphi) \vee \\ M \rightarrow m_{mod}(ve, \varphi, Mod)}} \exists\, VarsIn(ve):\ refinesEv\ ve \wedge \varphi\Big) \tag{2.3}$$

where the ILP definitions are assumed to contain pairwise mutually disjoint set of variables.

2.3.4. Composition and Mechanisms

The composition of a set of ILPs is defined by

$$M_{complete} \leftrightarrow M \wedge M_{default}. \tag{2.4}$$

In terms of the architecture described in Chapter 5, the policy decision point evaluates the instantiations of Equation 2.1 (which in part also occur in Equation 2.3), and the policy enforcement point implements the right hand sides of the instances of Equation 2.3 and the "allow" case—where an intended event is transformed into an actual event—in Equation 2.3.

The formula $\Box(M_{complete})$ defines the semantics of all combined ILPs in a system. It is possible that this leads to an inconsistent definition. This is the case, for instance, if one modifying ILP transforms an a into a b, and another modifying ILP transforms a b into an a. This problem of conflicting ILPs, formally expressed as inconsistency, and other formal analysis problems (e.g. does a certain implementation level policy actually enforce a given specification-level policy?) are not treated in this work. An approach to tackle the problem by leveraging model-checking technologies is discussed in [139].

ILPs and default rules can be implemented using runtime verification or complex event processing technology. While, strictly speaking, the composition of ILPs (i.e. the predicates in $M_{complete}$) configures mechanisms, for brevity's sake the term *mechanism* will be used in the following to refer to both aspects, the monitoring technology and the configuration ILPs. This also explains why ILPs are denoted by the letters M and m.

For the sake of conciseness, the above definition of ILPs slightly simplifies the definition in [132]: The definition from the literature allows for the specification of multiple subsequent events to be performed by modifying or executing ILPs.

2.3.5. Example

As an example, consider a security requirement stipulating that Alice can print a report, called $report.xls$, only if it has been previously approved by Bob. A respective specification-level policy would be

$$\neg(E(print \mapsto \{obj \mapsto report.xls\}) \ \underline{until} \ E(approve \mapsto \{obj \mapsto report.xls, by \mapsto Bob\}).$$

Such policy can be enforced by different kinds of ILPs, like

- an *inhibiting* ILP that makes sure the report is not printed before approval,

$$m_{inh}(print \mapsto \{obj \mapsto report.xls\}, \\ \boxdot(\neg E(approve \mapsto \{obj \mapsto report.xls, by \mapsto Bob\}\}) \ \underline{before}^{-} \ 1))$$

- a *modifying* ILP that adds a watermark to the printed report if it has not been approved,

$$m_{mod}(print \mapsto \{obj \mapsto report.xls\}, \\ \boxdot(\neg E(approve \mapsto \{obj \mapsto report.xls, by \mapsto Bob\}\})) \ \underline{before}^{-} \ 1, \\ \{print \mapsto \{obj \mapsto report.xls, watermark \mapsto \text{“unapproved”}\}\})$$

- or an *executing* ILP that triggers the automated notification of the administrator,

$$m_{exc}(print \mapsto \{obj \mapsto report.xls\},\\ \boxdot(\neg E(approve \mapsto \{obj \mapsto report.xls, by \mapsto Bob\}))\ \underline{before}^{-}\ 1,\\ \{notify \mapsto \{obj \mapsto report.xls, dst \mapsto admin, msg \mapsto \text{"unapproved printing"}\}\}).$$

2.4. Policy Activation and Violation

Defining the semantics of combined ILPs as $\Box(M_{complete})$, as suggested in Section 2.3.4, does not state from which point onwards this formula should hold. A simplistic approach is to assume that all ILPs are active from time point 0 onwards. In reality, with policy lifecycle management, for an ILP π with name $\overline{\pi}$, there would be activation and deactivation events parameterized by the name of the ILP and an abstract moment in time, n. The semantics of a ILP would then amount to

$$\Box(activate(\overline{\pi}, n) \implies \pi\ \underline{until}\ deactivate(\overline{\pi})),$$

starting at activation time n, and the operators of Φ^{-} would have to be augmented by a further parameter n and adjusted to "look back" only until $\max(0, n)$. Policy lifecycle management, however, is not the subject of this work, and this issue is not furtherly discussed here.

In the system described in this work, once a policy is violated, it remains violated forever, i.e. the violation is reported at every moment in time after the one in which the violation occurred. After reporting the violation once, however, other strategies could be applied, such as deactivating the policy, or resetting it to its initial state, or locking the complete system (i.e. inhibiting every possible event). Depending on the context, each of this strategies may or may not make sense. Because a correct answer does not exists in general, *policies themselves should state what happens after they have been violated for the first time*. If this information is not present, then the policy lifecycle management should rely on some default strategies. This problem, tightly connected to the policy lifecycle management issues, is not part of this work, (and thus intentionally ignored,) assuming the intervention of a security expert, after a reported violation, to restore the system to a safe state.

2.5. Detective and Preventive Enforcement

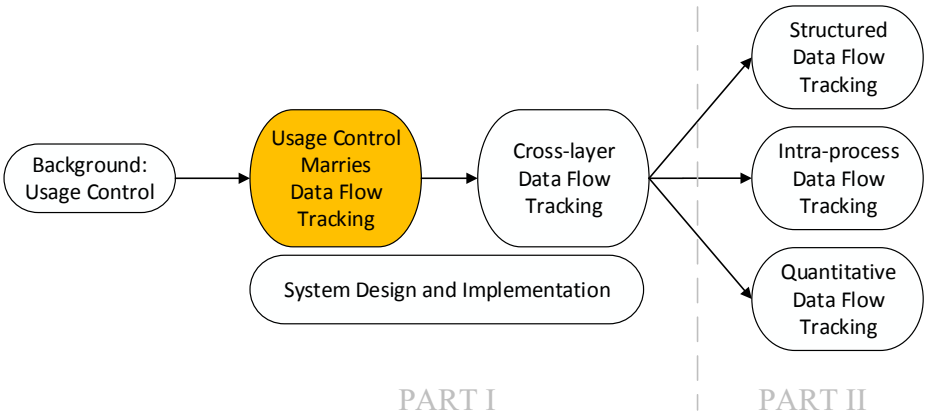
Inhibitors and modifiers are used for preventive enforcement: their goal is to ensure that a policy is adhered to. Executors can be used for both preventive and detective enforcement. An example for preventive enforcement has been presented in Section 2.3.5. Executors can be instrumental for detective enforcement as well: the event to be executed then is a mere logging event.

There also is a second possibility to implement detective enforcement that directly uses specification-level, i.e., future time, policies rather than ILPs. For safety properties—and all realistic usage control properties are safety properties because in realistic systems all duties are time-bounded—there exists an earliest moment in time when a policy is violated

or satisfied forever. The respective shortest "good" and "bad" prefixes can be precisely defined and detected at runtime [101, 79]. If the violation of a specification-level policy is detected by using this technology, then any kind of compensating action can be undertaken. These include undoing an action, lowering trust ratings, implementing penalties, etc. Formal details are provided elsewhere [81].

3. Usage Control Marries Data Flow Tracking

This chapter describes a generic model and language to augment the usage control concepts of Chapter 2 with data flow tracking features. This work is one of the core contributions of the thesis and has been published in [138]. The discussion on soundness is taken from [109], an unpublished work co-authored by the author of this dissertation.



As introduced in Chapter 1, *data* (e.g. a picture) exists within a system in form of multiple representations. These possibly reside at different layers of abstraction (e.g. a file at the operating system layer, a set of pixels at the windowing manager layer, a browser object at the application layer, etc.). Usage control requirements are often naturally expressed in terms of *data* ("*delete this picture*") rather than layer-specific representations ("*unlink* $file_1$, $file_2$,..., $file_n$, *empty clipboard, delete* $mail_{xyz}$ *from archive, erase* $backup_{abc}$, etc. "). In the remaining of this work, these representations are called *containers*.

In order to enforce policies related to data rather than containers (and thus events), a system must distinguish between the two, i.e. usage control decisions should take into account the dissemination of data among different containers. For instance, the requirement "*delete this picture*" should be enforced by deleting every container that contains this picture's data.

In the next sections, the usage control model and language described in the previous chapter are augmented with data flow tracking features, to track data dissemination at an arbitrary layer of abstraction (the generalization to data flows across layers is the goal of Chapter 4). The fundamental research questions addressed in this chapter are

How can usage control requirements be specified and enforced over all representations of the same data at once?

What does it mean that a certain data flow tracking solution is "sound"?

3.1. System model

Let systems be described as tuples

$$(\mathcal{D}, \mathcal{S}, \mathcal{C}, \mathcal{F}, \Sigma, \sigma_i, \mathcal{R})$$

where $\mathcal{D}$ is a set of data elements, $\mathcal{S}$ is the set of system events introduced in Section 2.1.3, $\mathcal{C}$ is a set of containers and $\mathcal{F}$ is a set of identifiers used to simplify model renaming activities.

States Σ are defined by three mappings:

- a *storage function* s of type $\mathcal{C} \to \mathbb{P}(\mathcal{D})$ that describes which set of data is potentially stored in which container;
- an *alias function* l of type $\mathcal{C} \to \mathbb{P}(\mathcal{C})$ that captures the fact that some containers may implicitly get updated whenever other containers do;
- a *naming function* f that provides names for containers and that is of type $\mathcal{F} \to \mathcal{C}$.

Therefore,

$$\Sigma = (\mathcal{C} \to \mathbb{P}(\mathcal{D})) \times (\mathcal{C} \to \mathbb{P}(\mathcal{C})) \times (\mathcal{F} \to \mathcal{C}).$$

$\sigma_i \in \Sigma$ is the initial state of the system in which any container is mapped to an empty set of data and has no name. The transition relation (or *monitor*)

$$\mathcal{R} \subseteq \Sigma \times \mathbb{P}(\mathcal{S}) \to \Sigma$$

is the core of the data flow tracking model, encoding how the execution of events affects the dissemination of data in the system. At runtime, the usage control infrastructure maintains the data state Σ. Events are intercepted, and the information state is updated according to $\mathcal{R}$. Note that $\mathcal{R}$ applied to a sequence of events is the recursive application of $\mathcal{R}$ to each event in the sequence (i.e. $\mathcal{R}(\sigma, \langle e_1, e_2, ..., e_n \rangle) = \mathcal{R}(\mathcal{R}(...\mathcal{R}(\sigma, e_1), e_2)...e_n)$.

As already mentioned in Section 2.1.3, events happening within the same timestep are assumed to be *independent* from each other, i.e. any possible serialization of them would result in the same state:

$$\forall \sigma \in \Sigma : \mathcal{R}(\sigma, \emptyset) = \sigma$$
$$\forall \sigma \in \Sigma; Es \subseteq \mathcal{S}; e \in \mathcal{S} : e \in Es \implies \mathcal{R}(\sigma, Es) = \mathcal{R}(\mathcal{R}(\sigma, \{e\}), Es \setminus \{e\})$$

Again, this assumption may sound restrictive, but in concrete implementations it is seldom the case that multiple events take place at the same time; when that is the case, it is usually possible to establish a clear order between events by observing additional parameters, e.g. system timestamp.

In the usage control model of Chapter 2, data is addressed by referring to specific representations of it as event parameters. For instance, the ILP described in Section 2.3.5 stipulates that if a file (a specific representation and a specific container) called *report.xls* has not been approved, it cannot be exported. The model described in this chapter addresses the situation where a copy of that file, *report2.xls*, should not be exported either.

To this end, the semantic model is extended by *data usages* that allow for the specification of protection requirements for all representations rather than just one. Using the data flow tracking model, the framework computes the data state of the system at each timestep n:

it considers the system trace until n, extracts the respective events in each step, iteratively computes the successor data state for each data state and eventually obtain the data state at time n. In an implementation, of course, the recursive computation is done using state machines to maintain the data state of the system at each moment in time (see Chapter 5).

3.2. Data, Containers, and Events

The framework needs to distinguish between *data* and *containers* for data. At the specification layer, this leads to the distinction between two classes of events according to the "type" of the *obj* parameter (see Section 2.1.1): events of class *dataUsage* define actions on data objects. The intuition is that these pertain to *every representation* of the data. In contrast, events of class *containerUsage* refer to one single container. In a real system, only events of class containerUsage can happen. This is because each monitored event in a trace targets a specific representation of the data (a file, a memory region, etc). dataUsage events are used only in the definition of policies, where it is possible to define a rule abstracting from the specific representation of a data item.

There also exists a third class of events, *noUsage* events, that do not define usages of containers or data. noUsage events include notifications, penalties, etc. The obj parameter of noUsage events is equal to $\emptyset$. Every event that is not a containerUsage nor a dataUsage event is a noUsage event. Let $getclass$ be the function that extracts the type of events.

$$
\begin{aligned}
&EventClass = \{dataUsage, containerUsage, noUsage\}\\
&getclass : \mathcal{E} \rightarrow EventClass\\
&\mathcal{D} \cup \mathcal{C} \subseteq PValue \quad \wedge \quad \mathcal{C} \cap \mathcal{D} = \emptyset
\end{aligned}
$$

The system has to satisfy the following conditions:

$$
\begin{aligned}
&\forall e \in \mathcal{E} : getclass(e) = dataUsage \iff \exists par \in \mathcal{D} : (obj \mapsto par) \in e.p\\
&\forall e \in \mathcal{E} : getclass(e) = containerUsage \iff \exists par \in \mathcal{C} : (obj \mapsto par) \in e.p.\\
&\forall e \in \mathcal{E} : getclass(e) = noUsage \iff (obj \mapsto \emptyset) \in e.p.
\end{aligned}
$$

Note that dataUsage events are similar to events with a variable obj parameter (see Section 2.1.1), where in this case the variable is quantified over every container that contains a certain data item. The fundamental difference between the two categories is that, in dataUsage events, the set $VVAl$ of possible values for variable obj is not fixed, but it may change at every moment in time, reflecting the changes in data dissemination.

Section 2.1.2 introduced relation $refinesEv$ and the idea of implicit quantification over unmentioned parameters when specifying events in policies. For example, if an obligation prohibits the event $print \mapsto \{obj \mapsto rep.xls\}$, then the event $print \mapsto \{obj \mapsto rep.xls, printer \mapsto HP - 001\}$ is prohibited as well.

This definition is now extended: an event of class dataUsage can be refined by an event of class containerUsage if the latter is related to a specific representation of the data the former refers to. As in the original definition, in both cases the more refined event can have more parameters than the more abstract event.

An event e_1 refines an event e_2 if and only if

- e_1 and e_2 both have the same class (containerUsage or dataUsage) and
- e_1 *refinesEv* e_2;

or

- if e_1 is a containerUsage and e_2 a dataUsage event,
- e_1 and e_2 have the same event name,
- in the current data state, there exists a data item *d* stored in a container *c* such that $(obj \mapsto c) \in e_1.p$ and $(obj \mapsto d) \in e_2.p$, and
- all parameters (except for *obj*) of e_2 have the same value in e_1, and e_1 may possibly have additional parameters.

Note that this in this definition, the data dissemination state Σ is needed in order to decide the refinement. Formally, relation

$$refinesEv_i \subseteq (\mathcal{E} \times \Sigma) \times \mathcal{E},$$

which checks whether one event e_1 refines another event e_2 also w.r.t. data and containers, is defined as

$$\begin{aligned}
&\forall e_1, e_2 \in \mathcal{E}, \forall \sigma \in \Sigma : (e_1, \sigma)\ refinesEv_i\ e_2 \iff \\
&\quad (getclass(e_1) = getclass(e_2) \wedge e_1\ refinesEv\ e_2) \vee \\
&\quad (getclass(e_1) = containerUsage \wedge getclass(e_2) = dataUsage \wedge \\
&\qquad e_1.n = e_2.n \wedge \exists d \in \mathcal{D}, \exists c \in \mathcal{C} :\ d \in \sigma.s(c) \wedge \\
&\qquad obj \mapsto c \in e_1.p \wedge obj \mapsto d \in e_2.p \wedge e_2.p \setminus \{obj \mapsto d\} \subseteq e_1.p \setminus \{obj \mapsto c\}).
\end{aligned}$$

where $\sigma.s$ denotes the storage function of state σ.

3.2.1. Computing the Data State

Function $states : (Trace \times \mathbb{N}) \to \Sigma$ is used to compute the information state σ at a given moment in time.

$$states(t, n) = \begin{cases} \sigma_i & \text{if } n = 0 \\ \mathcal{R}(states(t, n-1), t(n-1)) & \text{if } n > 0 \end{cases}$$

$states(t, n)$ represents the state of data dissemination after executing trace t until timestep $n - 1$ included, i.e. the state in which events at time n are executed. With the help of $refinesEv_i$ and *states*, it is possible to redefine the satisfaction relation for event expressions in the context of data and container usages by adding the data state as additional argument to $\models_e$, obtaining $\models_{e,i} \subseteq (\mathcal{S} \times \Sigma) \times \Psi$, defined as:

$$\begin{aligned}
&\forall e' \in maxRefEv, \forall e \in \mathcal{VE}, \forall \sigma \in \Sigma, \exists e'' \in \mathcal{E} : \\
&\qquad ((e', actual), \sigma) \models_{e,i} E(e) \iff (e', \sigma)\ refinesEv_i\ e'' \wedge e'' \in Inst_\varepsilon(e) \\
&\wedge \quad ((e', intended), \sigma) \models_{e,i} I(e) \iff (e', \sigma)\ refinesEv_i\ e'' \wedge e'' \in Inst_\varepsilon(e).
\end{aligned}$$

3.2.2. State-based Operators

In the semantic model, policies are defined on sequences of events. The idea of a policy is to describe certain situations to be avoided or enforced. However, in practice there usually is an almost infinite number of different sequences of events that lead to the same situation, e.g., the copy or the deletion of a file. Instead of listing all these sequences, it appears more convenient in situations of this kind to define a policy based on the description of the

(data flow state of the) system at that specific moment. To define such types of formulae, the model supports a new set of *state-based operators*, called Φ_s,

$$\Phi_s ::= \underline{isNotIn}(\mathcal{D}, \mathbb{P}(\mathcal{C}))|\underline{isCombinedWith}(\mathcal{D}, \mathcal{D})$$

plus the macro $\underline{isOnlyIn}(\mathcal{D}, \mathbb{P}(\mathcal{C}))$ with $\underline{isOnlyIn}(d, Cs) \Leftrightarrow \underline{isNotIn}(d, \mathcal{C} \setminus Cs)$.

Intuitively, $\underline{isNotIn}(d, Cs)$ is true if data d is not present in any of the containers in set Cs. This is useful to express constraints such as "*report.xls must not be distributed over the network*", which becomes $\Box(\underline{isNotIn}(report.xls, \{c_{net}\}))$ for a network container (i.e. any socket) c_{net}. $\underline{isCombinedWith}(d_1, d_2)$ checks if data items d_1 and d_2 are combined in one container. This is useful to express simple Chinese Wall policies. $\underline{isOnlyIn}$, the dual of $\underline{isNotIn}$, is used to express concepts such as "*data d has been deleted*" ($\underline{isOnlyIn}(d, \emptyset)$).

Leveraging the *states* function defined before, the semantics of the data usage operators in Φ_s can be defined in terms of event traces by $\models_s \subseteq (Trace \times \mathbb{N}) \times \Phi_s$:

$$\begin{aligned}
&\forall t \in Trace; n \in \mathbb{N}; \varphi \in \Phi_s; \sigma \in \Sigma : \ (t,n) \models_s \varphi \iff \sigma = states(t,n) \wedge \\
&\quad \exists d \in \mathcal{D}, Cs \subseteq \mathcal{C} : \varphi = \underline{isNotIn}(d, Cs) \wedge \forall c' \in \mathcal{C} : \\
&\qquad d \in \sigma.s(c') \implies c' \notin Cs \\
&\quad \vee \exists d_1, d_2 \in \mathcal{D} : \varphi = \underline{isCombinedWith}(d_1, d_2) \wedge \exists c' \in \mathcal{C} : \\
&\qquad d_1 \in \sigma.s(c') \wedge d_2 \in \sigma.s(c').
\end{aligned}$$

The language Φ can now be augmented with the state based operators in Φ_s, obtaining the new language Φ_i

$$\begin{aligned}
\Phi_i ::= \ & (\Phi_i)|\Psi|\underline{false}|\Phi_i \ \underline{implies} \ \Phi_i|\underline{forall} \ VName \ \underline{in} \ VVal : \Phi| \\
& \Phi_i \ \underline{until} \ \Phi_i|\Phi_i \ \underline{after} \ \mathbb{N}|\underline{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi)|\underline{repuntil}(\mathbb{N}, \Psi, \Phi_i) \\
& |\Phi_s
\end{aligned}$$

and its semantics $\models_{f,s} \subseteq (Trace \times \mathbb{N}) \times \Phi_i$

$$\begin{aligned}
&\forall t \in Trace; n \in \mathbb{N}; \varphi \in \Phi_i : (t,n) \models_{f,s} \varphi \Leftrightarrow \varphi \neq \underline{false} \wedge \\
&\quad \big(\exists e \in \mathcal{VE} : (\varphi = E(e) \vee \varphi = I(e)) \wedge \exists e' \in t(n) : e' \models_e \varphi \\
&\quad \vee \exists \psi, \chi \in \Phi_i : \varphi = \psi \ \underline{implies} \ \chi \wedge \neg((t,n) \models_f \psi) \vee (t,n) \models_{f,s} \chi \\
&\quad \vee \exists \gamma \in \Gamma : \varphi = eval(\gamma) \wedge [\![\varphi]\!]_{eval} = true \\
&\quad \vee \exists vn \in VName; vs \in VVal; \psi \in \Phi_i \ : \\
&\qquad \varphi = (\underline{forall} \ vn \ \underline{in} \ vs : \psi) \wedge \forall vv \in vs : (t,n) \models_{f,s} \psi[vn \mapsto vv] \\
&\quad \vee \exists \psi, \chi \in \Phi_i : \varphi = \psi \ \underline{until} \ \chi \wedge \big(\forall v \in \mathbb{N} : n \leq v \Rightarrow (t,v) \models_{f,s} \psi \\
&\qquad \vee \exists u \in \mathbb{N} : n < u \wedge (t,u) \models_{f,s} \chi \wedge \forall v \in \mathbb{N} : n \leq v < u \Rightarrow (t,v) \models_{f,s} \psi\big) \\
&\quad \vee \exists m \in \mathbb{N}; \psi \in \Phi_i : \varphi = \psi \ \underline{after} \ m \wedge (t, n+m) \models_{f,s} \psi \\
&\quad \vee \exists m \in \mathbb{N}_1; l, r \in \mathbb{N}; \psi \in \Psi : \varphi = \underline{replim}(m, l, r, \psi) \wedge \\
&\qquad l \leq \#\{j \in \mathbb{N}_1 | j \leq m \wedge (t, n+j) \models_{f,s} \psi\} \leq r \\
&\quad \vee \exists m \in \mathbb{N}; \psi \in \Psi, \chi \in \Phi_i : \varphi = \underline{repuntil}(m, \psi, \chi) \\
&\qquad \wedge \big((\exists u \in \mathbb{N}_1 : (t, n+u) \models_{f,s} \chi \wedge (\forall v \in \mathbb{N}_1 : v < u \Rightarrow \neg((t, n+v) \models_{f,s} \chi)) \\
&\qquad\quad \wedge (\#\{j \in \mathbb{N}_1 | j \leq u \wedge t(n+j) \models_{f,s} \psi\}) \leq m) \\
&\qquad\quad \vee (\#\{j \in \mathbb{N}_1 | t(n+j) \models_{f,s} \psi\}) \leq m\big)\big) \\
&\quad \vee \varphi \in \Phi_s \wedge (t,n) \models_s \varphi
\end{aligned}$$

Note that the definition of Φ_i and of its semantics are almost identical to those of Φ, except for the last line (highlighted in red), which includes the state based operators from Φ_s.

In a similar way, it is possible to define a past variant of the language (Φ_i^-) and its semantics $\models_{fs^-} \subseteq (Trace \times \mathbb{N}) \times \Phi_i^-$.

$$\begin{aligned}\Phi_i^- ::= \; & (\Phi_i^-) | \Psi | \underline{false} | \Phi_i^- \; \underline{implies}^- \; \Phi_i^- | \underline{forall}\; VName\; \underline{in}\; VVal : \Phi_i^- | \\ & \Phi_i^- \; \underline{since}^- \; \Phi_i^- | \Phi_i^- \; \underline{before}^- \; \mathbb{N} | \underline{replim}^-(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) | \underline{repsince}^-(\mathbb{N}, \Psi, \Phi_i^-) \\ & | \Phi_s\end{aligned}$$

$$\begin{aligned}
&\forall t \in Trace; n \in \mathbb{N}; \pi \in \Phi_i^- : (t,n) \models_{fs^-} \pi \Leftrightarrow (\pi \neq \underline{false}) \wedge \\
&\quad \big(\exists e \in \mathcal{VE} : (\pi = E(e) \vee \pi = I(e)) \wedge \exists e' \in t(n) : e' \models_e \pi \\
&\vee \quad \exists \psi, \chi \in \Phi_i^- : \pi = \psi \; \underline{implies}^- \; \chi \wedge \neg((t,n) \models_{fs^-} \psi) \vee (t,n) \models_{fs^-} \chi \\
&\vee \quad \exists \gamma \in \Gamma : \pi = eval(\gamma) \wedge [\![\pi]\!]_{eval} = true \\
&\vee \quad \exists vn \in VName; vs \in VVal; \psi \in \Phi_i^- : \\
&\qquad \pi = (\underline{forall}\; vn\; \underline{in}\; vs : \psi) \wedge \forall vv \in vs : (t,n) \models_{fs^-} \psi[vn \mapsto vv] \\
&\vee \quad \exists \psi, \chi \in \Phi_i^- : \pi = \psi \; \underline{since}^- \; \chi \wedge \big((\forall v \in \mathbb{N} : v \leq n \Rightarrow (t,v) \models_{fs^-} \psi) \\
&\qquad \vee (\exists u \in \mathbb{N} : u \leq n \wedge (t,u) \models_{fs^-} \chi \wedge \forall v \in \mathbb{N} : u < v \leq n \Rightarrow (t,v) \models_{fs^-} \psi)\big) \\
&\vee \quad \exists m \in \mathbb{N}; \psi \in \Phi_i^- : \pi = \psi \; \underline{before}^- \; m \wedge n \geq m \wedge (t, n-m) \models_{fs^-} \psi \\
&\vee \quad \exists m, l, r \in \mathbb{N}; \psi \in \Psi; : \varphi = \underline{replim}^-(m,l,r,\psi) \\
&\qquad \wedge \; l \leq (\#\{j \in \mathbb{N} | j \leq min(m,n) \wedge t(n-j) \models_{fs^-} \psi\}) \leq r \\
&\vee \quad \exists m \in \mathbb{N}; \psi \in \Psi; \chi \in \Phi; e \in \mathcal{E} : \varphi = \underline{repsince}^-(m, \psi, \chi) \\
&\qquad \wedge \big((\exists u \in \mathbb{N}_1 : n \geq u \wedge (t, n-u) \models_{fs^-} \chi \wedge (\forall v \in \mathbb{N} : v < u \Rightarrow \neg((t, n-v) \models_{fs^-} \chi)) \\
&\qquad\quad \wedge (\#\{j \in \mathbb{N} | j \leq u \wedge t(n-j) \models_{fs^-} \psi\} \leq m)) \\
&\qquad \vee (\#\{j \in \mathbb{N} | j \leq n \wedge t(n-j) \models_{fs^-} \psi\} \leq m)\big)\big) \\
&\vee \quad \varphi \in \Phi_s \wedge (t,n) \models_s \varphi
\end{aligned}$$

Leveraging $\models_{fs^-}$, ILP definitions of Section 2.3.2 can be augmented to allow for state-based operators in the conditions ($\varphi \in \Phi_i^-$). An example of a possible concrete syntax for the language is given in Appendix A.

3.3. Use Case (Single Layer)

As a proof-of-concept, the combined model described above has been instantiated for the application scenario described in Section 1.4. In this context, two instances of the model are considered: one for the Microsoft Excel application used by Alice to prepare her reports (`EX`) and one for the underlying Microsoft Windows (`OS`) operating system (in case of other operating systems, like OpenBSD, an additional monitor may be required for the windowing manager [136]).

> *Alice works as analyst for ACME Inc., a smartphone manufacturer. Her job consists in editing and preparing reports for suppliers and for other departments. Alice uses* ***Microsoft Excel*** *to prepare her reports and a third-party application, also used by all suppliers, to place the orders. In order to prevent accidental and intentional disclosures of sensitive data, information flow control mechanisms are in place at the application level, e.g.* ***preventing highly sensitive reports form being printed on***

> ***shared printers**, and at the operating system layer, e.g **preventing files that are tagged as confidential from begin sent outside the company premises and from appearing in a screenshot**.*

Note that requirements like "*This sheet of the workbook cannot be printed*" or "*No screenshot of this report can be taken*" can be enforced by tracking data at the single layers individually; examples of requirements that require cross-layer tracking of data can be found in Chapter 4. The second part of this work addresses the problem of how to enforce different kinds of requirements and/or the same kind of requirements in a more precise way.

3.3.1. Notation

The instantiation of the transition relation $\mathcal{R}$ for different layers of abstraction can be formalized in a simple way by introducing a function update notation. For a function $m : S \to T$ and an $s \in S$, let $m[s \leftarrow expr] = m'$ with $m' : S \to T$ such that

$$m'(y) = \begin{cases} expr & \text{if } y = s \\ m(y) & \text{otherwise} \end{cases}$$

This notation is also lifted to sets of changed domain elements. For a variable $x \to X$ with $X \subseteq S$, $m[x \leftarrow expr]_{x \in X} = m'$ with $m' : S \to T$ such that

$$m'(y) = \begin{cases} expr & \text{if } y \in X \\ m(y) & \text{otherwise} \end{cases}$$

This notation simplifies the description of the storage, alias, and naming functions updates in the data flow model e.g. to describe that an event adds a new mapping between a specific data item and a container. Multiple function updates on disjoint sets with simultaneous and atomic replacements can be defined within a single expression as follows:

$$m[x_1 \leftarrow expr_{x_1}; \ldots; x_n \leftarrow expr_{x_n}]_{x_1 \in X_1, \ldots, x_n \in X_n} = \\ \Big(\cdots\big((m[x_1 \leftarrow expr_{x_1}]_{x_1 \in X_1})[x_2 \leftarrow expr_{x_2}]_{x_2 \in X_2}\big)\cdots\Big)[x_n \leftarrow expr_{x_n}]_{x_n \in X_n}.$$

Besides this function update notation, the asterisk operator r^* is used to denote the transitive reflexive closure of a relation r. In particular, the notation $l^*(c)$ is used to retrieve the closure of the alias function l which effectively returns all containers c_i that are transitively referenced by container c.

For the sake of brevity, this section uses a simplified notation for system events: events, in the previous sections denoted by $\mathcal{E} \subseteq EName \times \mathbb{P}(PName \times PValue)$, in this section are modeled as projections of the corresponding API calls, system calls, etc.. The projection is done in a way that events only contain parameters that are relevant in terms of data flow. Events are then denoted by $\mathcal{E} := EName(PValue, \ldots, PValue)$, where the association between parameter values and names is implicitly given by their order.

Note that the monitors described in the next sections are specified in terms of single events; this is because, as discussed in Section 2.1.2, multiple events observed in the same timestep are independent. As such, they can be processed in a serialized fashion and result in the same final state regardless of the processing order. Also note that while in

Section 3.1 traces of the system are defined in terms of maximally refined events, the semantics for the monitors described in this section only mentions few relevant parameters for the events. This choice has been made to keep the notation simple and focused on the relevant parameters; the reader must be aware that an implicit universal quantification over the parameters that are not mentioned is always intended.

3.3.2. Operating System Layer

This example considers an instantiation of the model at the operating system layer for Microsoft Windows. This instantiation is based on the interception of function calls to the Windows API, the primary API of Windows, used by user-mode processes to access system resources and utilize extended system functionality like I/O or network communication. The choice of monitoring calls to the Windows API rather than calls to the native kernel API is motivated by the sparse documentation of the native API and its lack of high-level functionality, for e.g. user interface manipulation.

Conceptually, the Windows enforcement mediates the communication between user-mode processes and the operating system. Technically, the interception of calls to the Windows API is realized through function call interposition within the respective processes. More precisely, a subset of relevant Windows API calls is rerouted from user-mode processes to a custom detouring library that takes care of pre-processing and forwarding them to the evaluation infrastructure. Note, that this does not require any manual modification of the applications, thus supporting monitoring and control of any legacy Windows application that uses the standard Windows APIs.

To model the flow of data within this layer of abstraction, the data flow model described in Section 3.1is instantiated as follows: The set of data containers $\mathcal{C}$ is represented by entities that can contain data like windows, files, I/O devices, or the system clipboard:

$$\mathcal{C} := \mathcal{C}_{Windows} \cup \mathcal{C}_{Files} \cup \mathcal{C}_{Devices} \cup \mathcal{C}_{Clipboard} \cup \mathcal{C}_{M},$$

where $m_p \in \mathcal{C}_M$ refers to the container modeling the memory of process p. The set of names F covers all unique identifiers to these containers, i.e. windows ($F_{wHandle}$), processes (F_{pid}), file and device handles ($F_{fHandle} \cup F_{dHandle}$) and absolute file names (F_{fName}). Note that, in contrast to window handles, which are unique system-wide, different processes (e.g. with process ids "123" and "456") may associate the *same file handler* (e.g. "3") to different files ("file1" and "file2") or devices. For this reason, $\mathcal{F}$ is defined as

$$\mathcal{F} := F_{wHandle} \cup F_{fName} \cup F_{dName} \cup (F_{pid} \bowtie (F_{fHandle} \cup F_{dHandle}))$$

where $A \bowtie B$ indicates the set of all the single labels composed by the concatenation of an element in A and an element in B, separated by a colon[1]. In the same example, the labels "123:3" and "456:3" would be part of the domain of the naming function f, as well as "file1" and "file2", and $f(123{:}3) = f(\text{file1})$ and $f(456{:}3) = f(\text{file2})$.

Due to its enormous complexity, modeling the data flow behavior of the entire Windows API is daunting and not necessary to cover most data usage cases at this layer of abstraction. For this reason, the semantics of only a limited subset of events is presented here. A more comprehensive overview can be found in [164].

[1] To maintain a simple notation, the operator $\bowtie$ is overloaded to single elements such that $(A \bowtie B = C) \implies (a \bowtie b = c)$ for any $A = \{a\}, B = \{b\}$ and $C = \{c\}$.

Note that at the model level, the instantiation for Microsoft Windows does not fundamentally differ from the instantiation for any other operating system, like Android [55] or OpenBSD [73]. This is because each basic operating system event has a counterpart in any specific system, e.g. a *WriteFile* API call in Microsoft Windows and a *write* system call in a Unix system.

CreateFile Event

CreateFile is used to retrieve a handle to a (new or already existing) file, specified via its absolute file name. It thus creates a mapping between file name and handle.

$$\forall s : \mathcal{C} \to \mathbb{P}(\mathcal{D}), \forall l : \mathcal{C} \to \mathbb{P}(\mathcal{C}), \forall f : \mathcal{F} \to \mathcal{C}, \forall p \in F_{pid}, \forall fn \in \mathcal{F}_{fName}, \forall fh \in \mathcal{F}_{fHandle} : \\ \left((s, l, f), \mathit{CreateFile}\,(p, fh, fn), (s, l, f\,[(p \bowtie fh) \leftarrow f(fn)])\right) \in \mathcal{R}.$$

ReadFile Event

ReadFile induces a flow from a file to a process memory container. As in general it is not possible to know exactly which parts of a file are sensitive and where exactly in the process memory the sensitive parts flow to, a conservative estimation needs to be taken, assuming that all data flows into the process memory and all referenced containers (e.g. all windows belonging to the calling process).

$$\forall s : \mathcal{C} \to \mathbb{P}(\mathcal{D}), \forall l : \mathcal{C} \to \mathbb{P}(\mathcal{C}), \forall f : F \to \mathcal{C}, \forall p \in F_{pid}, \forall fh \in \mathcal{F}_{fHandle} : \\ \left((s, l, f), \mathit{ReadFile}\,(p, fh), (s[t \leftarrow s(f(p \bowtie fh)) \cup s(t)]_{t \in l^*(m_p), l, f})\right) \in \mathcal{R}$$

WriteFile Event

WriteFile results in a flow from a process memory container to a file. Here, a similar coarse-grained estimation of data flows is performed, as it is not possible to assess which parts of the process memory are written to the file.

$$\forall s : \mathcal{C} \to \mathbb{P}(\mathcal{D}), \forall l : \mathcal{C} \to \mathbb{P}(\mathcal{C}), \forall f : F \to \mathcal{C}, \forall p \in F_{pid}, \forall fh \in \mathcal{F}_{fHandle} : \\ \left((s, l, f), \mathit{WriteFile}\,(p, fh), (s[f(fn) \leftarrow s(f(fn)) \cup s(m_p)], l, f)\right) \in \mathcal{R}$$

3.3.3. Application Layer

The second instantiation of the generic model considered in the running example is for the Microsoft Excel spreadsheet application. At this level, the interception of data usage related events is performed through a custom Excel Add-in. Events in this context are triggered at the GUI level and cover tasks like opening a workbook, deleting a row, computing a value using data from different cells, creating a chart, or opening the print preview. Many events, e.g. **print**, are intercepted for usage control purposes ("*the content of this sheet cannot be printed*"), but their execution does not affect the dissemination of data in the model or, in case of events like **save**, such dissemination cannot be modeled in a non-trivial way

within Excel (e.g. flows of data from/to files). A model that captures data flows across different instantiations more precisely is described in Chapter 4.

In this domain, containers are the single cells ($\mathcal{C}_{Cell}$), whole worksheets ($\mathcal{C}_{Sh}$), or whole workbooks ($\mathcal{C}_{Wb}$), all the slots of the internal office clipboard ($\mathcal{C}_{OCB}$) and the printer (c_{Print}); additionally, a special container (c^U) to model the flow of data from/to files needs to be considered. This container is needed to model cross-layer events and its existence is motivated in detail in Section 3.4. Thus,

$$\mathcal{C} = \mathcal{C}_{Cell} \cup \mathcal{C}_{Sh} \cup \mathcal{C}_{Wb} \cup \mathcal{C}_{OCB} \cup \{c_{Print}, c^U\}$$

To identify these containers, the naming set $\mathcal{F}$ contains unique identifiers for each cells and each printer. Cell identifiers are of the standard form $(w, sh, r, c) \in WB\_PATHS \times SHEETS \times ROWS \times COLS$, where w is a possible path for a workbook, sh is the name of the worksheet in the given workbook and r and c identify, respectively, the row and the column of the cell. The special value $\emptyset$ indicates a universal quantification over that field, so while $(\mathit{file1.xls}, \mathit{Sheet1}, 5, 3)$ indicates the cell (5,3) of sheet $\mathit{Sheet1}$ in workbook $\mathit{file1.xls}$, $(\mathit{file1.xls}, \emptyset, \emptyset, \emptyset)$ is the label for the container that models the complete $\mathit{file1.xls}$ workbook, $(\mathit{file1.xls}, \mathit{Sheet1}, \emptyset, \emptyset)$ is the label for the container that models $\mathit{Sheet1}$, and $(\mathit{file1.xls}, \mathit{Sheet1}, 5, \emptyset)$ and $(\mathit{file1.xls}, \mathit{Sheet1}, \emptyset, 3)$ are the labels for, respectively, the complete row number 5 and the complete column number 3 of $\mathit{Sheet1}$ in $\mathit{file1.xls}$. Thus,

$$\mathcal{F} = WB\_PATHS \times SHEETS \times ROWS \times COLS$$

In this model, alias relations (see Chapter 3) are used to capture the fact that the value of a cell may depend on the values of other cells and to model the relation between a workbook, all the worksheet contained in it and all the cells contained in them. The event set $\mathcal{E}$ contains all Excel events that potentially lead to a flow of data between the specified containers.

Again, for brevity's sake, only the semantic of a limited subset of the modeled events is presented here. A more comprehensive overview can be found in [149] Note that this model only considers data flows within the spreadsheet application; as mentioned before, tracking data flows between Excel and the underlying operating system is out of the scope of this instantiation and is the goal of the model presented in Chapter 4.

Print Event

Print sends the content of the active workbook to the printer. Because it is possible to print only part of the workbook, the event is modeled in a generic way using the parameter $sel \subseteq \mathcal{F}$ to represent the set of selected cells to be printed. Notice that the data transferred to the printer is the content of the selected cells plus the content of every cell referenced by them, which is retrieved using the alias function l.

$$\forall s : \mathcal{C} \to \mathbb{P}(\mathcal{D}), \forall l : \mathcal{C} \to \mathbb{P}(\mathcal{C}), \forall f : \mathcal{F} \to \mathcal{C}, \forall sel \subseteq \mathcal{F} :$$
$$((s, l, f), Print(sel), (s[c_{Print} \leftarrow s(c_{Print}) \bigcup_{c \in \{c | \exists c' \in sel : c \in l^*(c')\}} s(c)], l, f)) \in \mathcal{R}$$

Erase Event

An erase event is observed when the content of a cell is deleted. In this case, the storage associated to the selected cell is deleted, and every alias to the cell is removed. As for the previous event, let $sel \subseteq \mathcal{F}$ be the set of selected cells.

$$\forall s : \mathcal{C} \to \mathbb{P}(\mathcal{D}), \forall l : \mathcal{C} \to \mathbb{P}(\mathcal{C}), \forall f : \mathcal{F} \to \mathcal{C}, \forall sel \subseteq \mathcal{F} :$$
$$((s, l, f), Erase(sel),$$
$$(s[f(n) \leftarrow \emptyset]_{\forall n \in sel}, l[x \leftarrow l(x) \setminus \{f(n)\}]_{\substack{\forall x \in \{x \mid \exists n \in sel : x \in l^*(n)\} \\ \forall n \in sel}}, f)) \in \mathcal{R}$$

Sheet Rename Event

When a sheet is renamed, all the references to the cells of that sheet need to be updated. Such update is modeled using the information provided by this event, i.e. the name of the workbook in which the sheet is located (wb), the old name (old) and the new name (new) of the sheet.

$$\forall s : \mathcal{C} \to \mathbb{P}(\mathcal{D}), \forall l : \mathcal{C} \to \mathbb{P}(\mathcal{C}), \forall f : \mathcal{F} \to \mathcal{C} :$$
$$((s, l, f), SheetRename(wb, old, new),$$
$$(s, l, f[(wb, new, r, c) \leftarrow f(wb, old, r, c); (wb, old, r, c) \leftarrow \emptyset]_{\substack{\forall r \in ROWS \\ \forall c \in COLS}}, f)) \in \mathcal{R}$$

3.4. Soundness (Single Layer)

This section formalizes a definition of *soundness* for data flow tracking at a generic layer of abstraction. This definition is then used as a basis to justify the soundness of the cross-layer models described in Chapter 4.

Note that, for simplicity's sake, the discussion of soundness in the remaining of this chapter and in the next one will assume the state of the system to be given by a storage function only. For this reason, the storage of a container c in state σ can be denoted as $\sigma(c)$. Integrating alias and naming function would only make the formalization more complex without introducing any additional fundamental challenge; for this reason, aliases and names are ignored in this section.

The model described so far can be instantiated at an arbitrary layer of abstraction, e.g. operating system, a data base, a windowing system, an application, etc. Let A be such layer. The goal of this section is to formalize desirable properties by relating the model for A to a very low level model $\perp$ with intuitive completeness and correctness properties. One could think at $\perp$ as the level of the CPU and volatile as well as persistent memory cells, representing the *real* execution of the system. This layer provides a notion of *value* of the containers $\mathcal{V} : \mathcal{C} \to \mathbb{N}$ that indicates the actual value stored in memory, and a trace execution semantics that changes the value of containers $\mathsf{eval} : \mathcal{V} \times seq(\mathcal{S}_\perp) \to \mathcal{V}$, such that the system at $\perp$ is given by $(\mathcal{D}_\perp, \mathcal{S}_\perp, \mathcal{C}_\perp, \Sigma_\perp, \sigma_i, \mathcal{R}_\perp, \mathcal{V}, \mathsf{eval})$.

A in contrast is some distinct higher layer. Set $\mathcal{L}$ denotes the set of all these high layers, while $\mathcal{L}^\perp = \mathcal{L} \cup \{\perp\}$. For $\dagger \in \mathcal{L}$, $\mathcal{C}_\dagger$ denotes the set of containers at layer $\dagger$. Note that the set of data $\mathcal{D}$ is layer-independent. Any layer A is related to $\perp$ by a pair of functions γ

and α that associate events and containers as follows: The idea is that an A-layer container corresponds to a set of $\bot$-layer containers (volatile and persistent memory cells) and an A-layer action to a sequence of *CPU*-level instructions (machine instructions such as `MOV`, `BNE`, `ADD`, `LEQ`). For a layer $\dagger \in \mathcal{L}^{\bot}$, each state $\sigma_\dagger \in \Sigma_\dagger$ is defined by the respective storage component (see discussion at the beginning of this section).

Relating states and events The model presented in Chapter 2 defines traces as sequences of sets of events. For simplicity's sake, the discussion on soundness presented here and in the next chapter will assume traces to be given by sequences of events. This assumption does not introduce any fundamental restriction, because given a trace in terms of sets of events t, it is always possible to determine a sequence of events s such that $\mathcal{R}(\sigma, t) = \mathcal{R}(\sigma, s)$, thanks to the assumption of independence of events within the same timestep (see Section 2.1.3).

Events at any layer are assumed to be unique and to contain an implicit timestamp and duration, yielding a natural order on a trace's events. Each event at a higher layer corresponds to a sequence of CPU-level instructions.

The discussion of soundness in this section and in the next chapter assumes it is possible to bijectively map an abstract sequence of events to a concrete sequence of events. This embodies the fundamental assumption of a *single-core system*: all traces can be uniquely sequentialized. Note that, in this sense, the concretization and abstraction functions are *ideal*: they deterministically relate ordered set of unique events with timestamps. This corresponds to the intuition that they relate to "what has really happened" in a monitored system, where for instance scheduling of concurrent processes has already been fixed.

The goal will be thus to reason about what has happened at the lowest layer, without ever actually having to monitor it, by assuming partial information on these abstraction/-concretization functions.

Moreover, *events at $\bot$ that do not correspond to an event at a higher layer are deliberately ignored*, e.g. those generated by an application for which there is no explicit monitor. The implication is that this usage control framework approach can only be sound w.r.t. those CPU-layer instructions for which a monitor at some layer exists.

In the following, abstraction and concretization functions for events and states are defined. For this purpose, α and γ are overloaded as follows. Let $\dagger \in \mathcal{L}$:

$$\begin{array}{lll} \textit{Events}: & \gamma_\dagger : seq(\mathcal{S}_\dagger) \to seq(\mathcal{S}_\bot), & \alpha_\dagger : seq(\mathcal{S}_\bot) \to seq(\mathcal{S}_\dagger) \\ \textit{States}: & \gamma_\dagger : \Sigma_\dagger \to \Sigma_\bot, & \alpha_\dagger : \Sigma_\bot \to \Sigma_\dagger \\ \textit{Containers}: & \gamma_\dagger : \mathcal{C}_\dagger \to \mathbb{P}(\mathcal{C}_\bot), & \alpha_\dagger : \mathcal{C}_\bot \to \mathbb{P}(\mathcal{C}_\dagger). \end{array}$$

such that

$$\begin{aligned} \gamma_\dagger(\sigma_\dagger) &= \left\{(c_\bot, \sigma_\dagger(c_\dagger)) : c_\dagger \in \mathsf{dom}(\sigma_\dagger) \wedge c_\bot \in \gamma_\dagger(c_\dagger)\right\} \\ \alpha_\dagger(\sigma_\bot) &= \{(c_\dagger, \sigma_\bot(c_\bot)) : c_\bot \in \mathsf{dom}(\sigma_\bot) \wedge c_\dagger \in \alpha_\dagger(c_\bot)\}. \end{aligned}$$

Additionally, $\forall \mathcal{C} \subseteq \mathcal{C}_\dagger : \gamma_\dagger(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \gamma_\dagger(c)$ and $\forall \mathcal{C} \subseteq \mathcal{C}_\bot : \alpha_\dagger(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \alpha_\dagger(c)$.

At each layer $\dagger \in \mathcal{L}$, there exists a special container $c_\dagger^U$ that represents the abstraction of all those $\bot$-layer containers that are not observable within $\dagger$ ($\forall c_\bot \in \mathcal{C}_\bot : (\forall c_\dagger \in \mathcal{C}_\dagger \setminus \{c_\dagger^U\} : \alpha_\dagger(c_\bot) \neq c_\dagger) \implies (\alpha_\dagger(c_\bot) = c_\dagger^U)$). By definition $\sigma(c_\dagger^U) = \mathcal{D}$ for any state σ, because $\mathcal{D}$, i.e. *all data*, is a conservative estimation for the content of an unknown container.

Given two states σ_1 and σ_2 at the same abstraction layer †, let $\sigma_1 \bowtie_\sigma \sigma_2$ denote the state at the same layer given by the union of the respective storage function.

$$\sigma_1 \bowtie_\sigma \sigma_2 = \{(c, D) | c \in \mathcal{C}_\dagger \wedge D = \sigma_1(c) \cup \sigma_2(c)\}$$

Recall that events at any layer are assumed to be unique and to contain an implicit timestamp, yielding a natural order on a trace's events. The ordered trace consisting of unique elements present in t_1 and t_2, traces of events at layer †, is denoted as $t_1 \bowtie_t t_2$.

Strategy In the following, the notion of taint propagation at the lowest abstraction layer ($\bot$) is related with that of *weak secrecy*. An ideal monitor $\mathcal{R}^\#_\bot$, sound with respect to weak secrecy, is defined in Section 3.4.1. This monitor is then used to define a notion of *soundness* (with respect to $\bot$) for a single layer A, which will then be used in Chapter 4 to justify soundness of the cross-layer data flow tracking.

3.4.1. Security Property at the $\bot$ Layer

Data flow tracking estimates which containers are "dependent" from the data stored in some other containers after a system run. The strongest guarantees in this sense are given by Non-Interference [67], which relates dependency of inputs and outputs in terms of pairs of executions (or state of variables before and after executing a program [159]). In the context of this work, Non-Interference can be defined as:

Definition 3.1 (Non-Interference). *Let $\mathcal{C}^i_H, \mathcal{C}^o_H \subseteq \mathcal{C}_\bot$ be sets of containers at $\bot$. A trace $t_\bot \in seq(\mathcal{S}_\bot)$ respects* Non-Interference *w.r.t. $\mathcal{C}^i_H, \mathcal{C}^o_H$ if*

$$\forall v, v' \in \mathcal{V} : \bigwedge_{c \in \mathcal{C}_\bot \setminus \mathcal{C}^i_H} v(c) = v'(c) \Longrightarrow \bigwedge_{c \in \mathcal{C}_\bot \setminus \mathcal{C}^o_H} \mathsf{eval}(v, t_\bot)(c) = \mathsf{eval}(v', t_\bot)(c)$$

In other words, the values of a certain memory region (represented by *low*) containers are independent from its complement (the *high* containers) after execution of a trace at $\bot$. This represents the notion of *absence of flows* from high to low containers. Monitoring Non-Interference is unfeasible [158] because it is a property over sets of traces (hyperproperty). Nevertheless, the goal of this work is to monitor data flows in single system runs. A relaxed notion of Non-Interference which can be monitored is formally captured by *weak-secrecy* [158].

Weak-secrecy relates a certain execution branch of a program with all other execution branches that exhibit the same behavior in terms of direct flows. To formally define this property in the context of this work, consider a trace $t_\bot \in seq(\mathcal{S}_\bot)$. The *branch-free* version of $t_\bot$, denoted as $bf(t_\bot)$, consists of the same assembly-level instructions in the same order, except for branch statements such as `BNE`, which are removed from the observed trace.

Definition 3.2 (Weak secrecy). *Let $\mathcal{C}^i_H, \mathcal{C}^o_H \subseteq \mathcal{C}_\bot$ be sets of containers at $\bot$. A trace $t_\bot \in seq(\mathcal{S}_\bot)$ respects* weak-secrecy *w.r.t. $\mathcal{C}^i_H, \mathcal{C}^o_H$ if its branch-free version $bf(t)$ is Non-Interferent w.r.t. $\mathcal{C}^i_H$,$\mathcal{C}^o_H$.*

$\mathcal{R}_\bot$ is the monitor that propagates labels in-between containers at layer $\bot$ as consequence of the execution of a trace.

Definition 3.3. *A monitor $\mathcal{R}_\perp$ is sound w.r.t. weak-secrecy if given an initial state σ_i, for all data items $d \in \mathcal{D}$, all traces $t_\perp \in seq(\mathcal{S}_\perp)$ respect weak-secrecy for the initial partition of the containers as induced by d: $\mathcal{C}_H^i = \{c \in \mathcal{C}_\perp \mid d \in \sigma_i(c)\}$ and, at the end of trace $t_\perp$, the resulting partition of the containers as computed by the monitor: $\mathcal{C}_H^o = \{c \in \mathcal{C}_\perp \mid d \in \mathcal{R}_\perp(\sigma_i, t_\perp)(c)\}$.*

In other words, if $\mathcal{R}_\perp$ claims a container c does not hold data d after the execution of a trace, then the values of c are independent from the values of d in the weak-secrecy sense. In the remaining of this section and in the respective argument for multi-layer systems in Chapter 4, $\mathcal{R}_\perp^\#$ is assumed to be the most precise sound monitor, i.e. for all $d \in \mathcal{D}$ and all traces $t_\perp \in seq(\mathcal{S}_\perp)$, the output partition $\mathcal{C}_H^o$ induced by any sound monitor includes the $\mathcal{C}_H^o$ partition induced by $\mathcal{R}_\perp^\#$.

3.4.2. Sources and Destinations

In practical terms, from the point of view of $\mathcal{R}_\perp^\#$, events move data from a containers to another: an instruction typically reads from a certain memory region and writes to another. For any given event e and a transition function $\mathcal{R}_\perp^\#$, let the functions $\mathbb{S}_{\mathcal{R}_\perp^\#} : \mathcal{S}_\perp \to 2_\perp^\mathcal{C}$ and $\mathbb{D}_{\mathcal{R}_\perp^\#} : \mathcal{S}_\perp \to 2_\perp^\mathcal{C}$ denote, respectively, the set of source and the set of destination containers of the events. These two functions are assumed to be given as an oracle of the event, such that the following property holds:

$$\forall \sigma, \forall c \in \mathcal{C}_\perp, \forall d \in \mathcal{D} : \\ d \in \mathcal{R}_\perp^\#(\sigma, e)(c) \implies d \in \sigma(c) \vee (\exists c' \in \mathbb{S}_{\mathcal{R}_\perp^\#}(e) : d \in c' \wedge c \in \mathbb{D}_{\mathcal{R}_\perp^\#}(e)).$$

In other words, if after executing e a certain container c contains data d, then d was already present in c before the execution of e, or there was a flow from a container in the sources of e to c, making c a destination.

Note that different partitions may fulfill this property. In the following, the oracle is assumed to provide the most precise ones, i.e. such that e respects weak secrecy w.r.t. $\mathbb{S}_{\mathcal{R}_\perp^\#}$, $\mathcal{C}_\perp \setminus \mathbb{D}_{\mathcal{R}_\perp^\#}$, and w.r.t $\mathcal{C}_\perp \setminus \mathbb{S}_{\mathcal{R}_\perp^\#}, \mathbb{D}_{\mathcal{R}_\perp^\#}$. In other words, there is Non-Interference between the partitions induced by sources and destinations, as depicted in Figure 3.1. Intuitively, this ensures sufficient "precision" of the source and destination functions: all relevant sources and all relevant destinations are captured. For any layer $\dagger \in \mathcal{L}$, let $\mathbb{S}_{\mathcal{R}_\dagger}$ and $\mathbb{D}_{\mathcal{R}_\dagger}$ represent the set of source and destination containers in $\mathcal{C}_\dagger$ such that the same relation also holds between $\mathcal{R}_\dagger$, $\mathcal{C}_\dagger$ and data.

For simplicity's sake, the notation of $\mathbb{S}$ and $\mathbb{D}$ is overloaded for traces of events $t \in seq(\mathcal{S}_\dagger)$ as $\mathbb{S}_{\mathcal{R}_\dagger}(t) = \bigcup_{e \in t} \mathbb{S}_{\mathcal{R}_\dagger}(e)$ and $\mathbb{D}_{\mathcal{R}_\dagger}(t) = \bigcup_{e \in t} \mathbb{D}_{\mathcal{R}_\dagger}(e)$. A similar overloading also applies to sets of events.

3.4.3. Single Layer Soundness

A state of the system $\sigma_A \in \Sigma_A$ is sound if, for every container $c_A \in \mathcal{C}_A$, the estimation of the data stored in c_A is a superset of (a sound estimation of) the data "actually" stored in it, i.e. of the data stored in the concretization of c_A. For this reason, soundness is defined w.r.t. a $\perp$-state and to a fixed pair of concretization/abstraction functions γ_A/α_A.

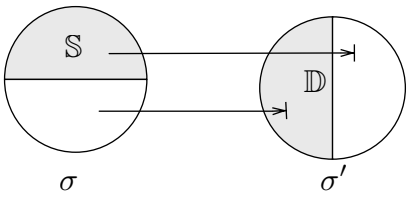


Figure 3.1.: Semantic property of source and destination partitions of a trace of events [109]. Circles represent the same whole memory; sources and destinations may hence overlap. Arrows indicate Non-Interference between respective memory regions before (σ) and after (σ') the execution of a trace of events.

Definition 3.4. *A state σ_A is* sound w.r.t. *$\sigma_\perp$, written $\sigma_\perp \vdash \sigma_A$, if and only if*

$$\forall c_A \in \mathcal{C}_A : \sigma(c_A) \supseteq \bigcup_{c_\perp \in \gamma_A(c_A)} \sigma_\perp(c_\perp).$$

This implies that $\forall \sigma_A \in \Sigma_A : \gamma_A(\sigma_A) \vdash \sigma_A$ and that $\forall \sigma_\perp \in \Sigma_\perp : \sigma_\perp \vdash \alpha_A(\sigma_\perp)$.

The data flow analysis for A is sound w.r.t. $\perp$ if the transition relation $\mathcal{R}_A$ preserves the soundness of the state (w.r.t. the canonical $\mathcal{R}^\#_\perp$ of Definition 3.3), i.e. if, given σ_A sound w.r.t. $\sigma_\perp$, $\forall e_A \in \mathcal{S}_A : \mathcal{R}^\#_\perp(\sigma_\perp, \gamma_A(e_A)) \vdash \mathcal{R}_A(\sigma_A, e_A)$.

The analysis for A is sound w.r.t $\perp$ if the initial state of the system σ^i_A is sound w.r.t. $\sigma^i_\perp$ and if $\mathcal{R}_A$ is sound w.r.t. $\mathcal{R}^\#_\perp$. Note that the definition of soundness is relative to flows observable at the $\perp$-layer and, necessarily, to a given concretization function.

Definition 3.5. *A monitor $\mathcal{R}_A$ at a layer A is sound w.r.t. $\perp$, written $\mathcal{R}^\#_\perp \vdash \mathcal{R}_A$, if given an initial state $\sigma^i_\perp \vdash \sigma^i_A$, modeling any trace of events $t_A \in seq(\mathcal{S}_A)$ results in a state σ_A which is sound with respect to the state reached by the canonical $\mathcal{R}^\#_\perp$ at $\perp$ for $\gamma_A(t_A)$. Formally,*

$$\begin{aligned}&\forall t_A \in seq(\mathcal{S}_A), \sigma^i_A \in \Sigma_A, \sigma^i_\perp \in \Sigma_\perp : \\ &\qquad \mathcal{R}^\#_\perp \vdash \mathcal{R}_A \iff \sigma^i_\perp \vdash \sigma^i_A \wedge \mathcal{R}^\#_\perp(\sigma^i_\perp, \gamma_A(t_A)) \vdash \mathcal{R}_A(\sigma^i_A, t_A)\end{aligned}$$

Lemma 3.6. *If $\mathcal{R}^\#_\perp \vdash \mathcal{R}_A$ for some $\mathcal{R}_A$ then for all $t_A \in seq(\mathcal{S}_A)$, the corresponding $\gamma_A(t_A) \in seq(\mathcal{S}_\perp)$ respects weak-secrecy for the partitions induced by $d \in \mathcal{D}$: $\mathcal{C}^i_H = \gamma_A(\{c \in \mathcal{C}_A \mid d \in \sigma_i(c)\})$ and the resulting partition of the containers as computed by the monitor at A: $\mathcal{C}^o_H = \gamma_A(\{c \in \mathcal{C}_A \mid d \in \mathcal{R}_A(\sigma_i, t_A)(c)\})$.*

This follows directly from the definition of soundness of a monitor and of weak-secrecy, because the partitions induced by d at A are supersets of the corresponding partitions at $\perp$. Note that also as direct corollary of soundness at A it follows that

$$\begin{aligned}\mathbb{S}_{\mathcal{R}^\#_\perp}(\gamma_A(e)) &\subseteq \gamma_A(\mathbb{S}_{\mathcal{R}_\dagger}(e)) \\ \mathbb{D}_{\mathcal{R}^\#_\perp}(\gamma_A(e)) &\subseteq \gamma_A(\mathbb{D}_{\mathcal{R}_\dagger}(e)).\end{aligned}$$

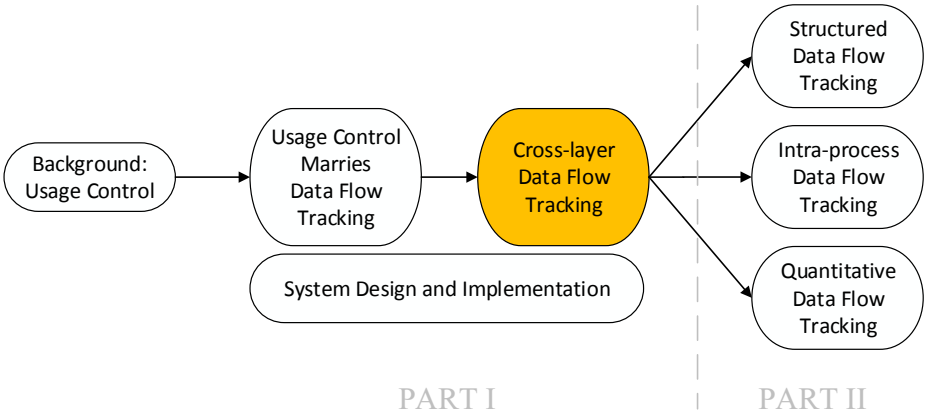
3.5. Conclusions

The work described in this chapter describes a language and a model to specify and enforce usage control requirements over all the representations of the same data at once. The framework is built on top of the usage control model described in the previous chapter and the different representations are identified by mean of a data flow tracking monitor that models the propagation of data in correspondence of system events' execution. The definition of the language and of the system model for concrete *data-centric usage control* answers the first research question presented at the beginning of the chapter and represents the first major contribution of this thesis.

Additionally, Section 3.4 formalizes the notion of soundness for a generic data flow tracking system, which answers the second research question and provides the basic notation and definitions for the discussion about soundness of the results presented in the next chapter.

4. Cross-layer Data Flow Tracking

This chapter describes a generic model to track flows of data across different instances of the model described in Chapter 3. This work represents the second major contribution of the thesis and is part of two unpublished works co-authored by the author of this dissertation [109, 137].



Section 3.4 presented a notion of soundness for data flow tracking at one particular layer of abstraction with respect to the lowest abstraction layer $\perp$. Such definition is based on the same idea of the relaxed variant of Non-Interference called *weak secrecy*. The goal of this chapter is to leverage these concepts in order to (1) define soundness for data flow tracking in the presence of multiple monitors for different layers of abstraction, (2) provide an algorithm that operationalizes the construction of a model to track flows of data across the different monitors and (3) prove the soundness of such algorithm. The remaining of this chapter is structured in four sections:

- Section 4.1 introduces two running examples, based on the instantiations described in Section 3.3, that are used to motivate the design choices;
- Section 4.2 formalizes the notion of soundness for a monitor in a composed system, defines $\hat{\mathcal{R}}_{A\otimes B}$, a simple monitor for a system composed by two layers of abstraction A and B and prove its soundness;
- Section 4.3 shows an example where additional information about A and B can lead to a more precise cross-layer tracking and presents a composed monitor $\dot{\mathcal{R}}_{A\otimes B}$ to perform it. This section also contains a formal argument for the soundness of $\dot{\mathcal{R}}_{A\otimes B}$, which constitutes the major result of this chapter and one of the main contributions of this thesis;
- Section 4.4 describe in detail an instantiation of $\dot{\mathcal{R}}_{A\otimes B}$ for the running example.

Note that for simplicity's sake, this work assumes a system composed by only two layers A and B; if the system is composed by $n > 2$ layers, then the same approach described in these pages can be applied recursively considering the $n-$th layer as A and the composition of the other $n - 1$ layers as B.

The fundamental research questions addressed in this chapter are

How can the relation between data representations and events at different layers be modeled?

What determines flows of data across different layers of abstraction?

What does it mean that a cross-layer data flow tracking solution is "sound"?

4.1. Motivating Example

> *Alice works as analyst for ACME Inc., a smartphone manufacturer. Her job consists in editing and preparing reports for suppliers and for other departments. Alice uses **Microsoft Excel** to prepare her reports and a third-party application, also used by all suppliers, to place the orders. In order to prevent accidental and intentional disclosures of sensitive data, data flow control mechanisms are in place at the **operating system layer**, e.g preventing files that are tagged as confidential from being sent outside the company premises or printed. The problem for Alice is that once she **loads a file** that is protected by a policy, e.g. "do not print", the same restriction applies to every further file she subsequently opens or **saves**.*

The running examples in this work are based on the two instantiations of the model described in Section 3.3: `EX` for Microsoft Excel, and `OS` for its underlying Microsoft Windows operating system. While the remaining of this chapter will focus on Excel and Windows, however, the events considered at the two layers will be *generic*, i.e. events that have a counterparts in any other operating system or application, like `WRITE()`, which corresponds to a `WriteFile` API call in Microsoft Windows and to a `Write` system call in a Unix system, or loading and saving a file from/to the disk.

4.1.1. File loading

The first example, depicted in Figure 4.1 is the typical operation of loading a workbook w from file $f.xls$ stored on the hard disk. This results in the event `LOAD`$(w, f.xls)$ at layer `EX` and in a sequence of `READ`(fd, pid) system calls at layer `OS`, preceded by a `OPEN`$(f.xls, fd, pid)$ system call to open the file handler and followed by a `CLOSE`(fd, pid) to close it, where pid is the process id of the Excel instance. If the container representing file $f.xls$ at the `OS` layer contained a certain data d, one would expect that after the load, also the container representing workbook w at the `EX` layer contains d. In other words, in

order to be intuitively *sound*[1], the system should reflect the fact that data d flowed from the file container at layer OS to the workbook container at layer EX.

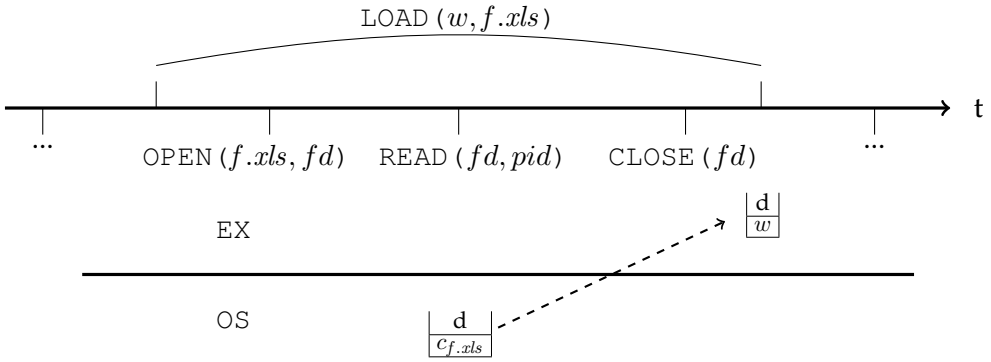


Figure 4.1.: Example of cross-layer flow of data d generated by loading a file in Excel [137].

4.1.2. File saving

The second example is the dual of the previous operation: saving the workbook w to file $f.xls$. It corresponds to a SAVE$(w, f.xls)$ event at the EX layer and to a sequence of WRITE(fd, pid) system calls at the OS layer (enclosed by an OPEN$(f.xls, fd, pid)$ and a CLOSE(fd, pid) events like in the File loading example). In this scenario, the flow of data d from the workbook in Excel to the file at the OS layer could be completely modeled at the OS layer: every data previously accessed by (i.e. stored in the container of) Excel process (from now on denoted as m_{app}) is then stored in the container representing the file. Such overapproximation is introduced also when the workbook being saved contains no sensitive data: at the OS layer no information about the sensitiveness of the content saved into the file is available and thus a conservative approach is taken. In a multi-layer context, this can be improved by having the data flow tracking monitor at the EX layer notifying the monitor at the OS layer about the data being saved (and implicitly, about the data *not* being saved), so the WRITE() system calls can be modeled precisely, mitigating the overapproximation issue.

4.2. A Sound Cross-layer Monitor

This section defines a notion of layer composition and discusses possible ways in which events observable at one layer may interfere with another layer. It then shows a first overly-conservative way to model composition and proves its soundness.

4.2.1. Soundness (Multi-layer)

Without loss of generality, it is safe to assume that $\mathcal{C}_A \cap \mathcal{C}_B = \emptyset$ for each pair of distinct $A, B \in \mathcal{L}^{\perp}$. Given two sound models for two layers of abstractions A and B in a system,

[1] a formalization of soundness is provided in Section 4.2.1

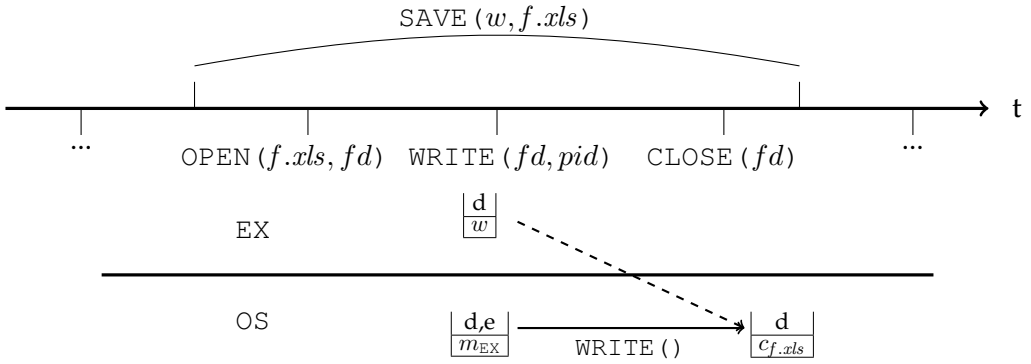


Figure 4.2.: Example of cross-layer flow that refines existing flows [137]. Note that if OS layer were considered in isolation, $c_{f.xls}$ would contain both data d and e

the goal is to define a sound model for the system composed by A and B, denoted $A \otimes B$.

The composed system is defined using the abstraction and concretization functions to compose the observations of monitors at the single layers. Let $\mathcal{C}_{A\otimes B} = \mathcal{C}_A \cup \mathcal{C}_B$ be the set of of containers in the composed system and $\mathcal{T}_{A\otimes B} \subseteq seq(\mathcal{S}_A) \times seq(\mathcal{S}_B)$ the set of event traces, given by pairs of traces observed in A and B. The composed state $\sigma_{A\otimes B} \in \Sigma_{A\otimes B} \subseteq \Sigma_A \times \Sigma_B$ is given as a pair of states in layers A and B respectively.

The notion of state soundness for single layers given in Definition 3.4 can then be extended to composed states as follows:

Definition 4.1 (Soundness Composed State)**.** *A state $\sigma_{A\otimes B}$ is sound w.r.t to a state $\sigma_\perp$, written $\sigma_\perp \vdash \sigma_{A\otimes B}$, if both components of the state are sound w.r.t. $\sigma_\perp$, i.e.*

$$\sigma_\perp \vdash \sigma_{A\otimes B} \iff \sigma_\perp \vdash (\sigma_{A\otimes B})|_A \wedge \sigma_\perp \vdash (\sigma_{A\otimes B})|_B$$

where the notation $|_l$ denotes the projection of the state to layer l.

In the following, the notation $|_l$ will also be used to denote the projection of traces of events to layer l. Given this definition of states and traces of events, it is possible to derive an ideal (w.r.t. $\perp$) composed monitor given by concretization and abstraction functions.

From a mathematical point of view, it is simple to compose two monitors as follows.

Definition 4.2 (Ideal composed monitor)**.** *Let t_A and t_B be traces at layers A and B, respectively, and σ_A and σ_B initial sound states. Let $\sigma^\perp_{A\otimes B} = \gamma_A(\sigma_A) \bowtie_\sigma \gamma_B(\sigma_B)$, $t^\perp_{A\otimes B} = \gamma_A(t_A) \bowtie_t \gamma_B(t_B)$ and $\sigma'^\perp_{A\otimes B} = \mathcal{R}^\#_\perp(\sigma^\perp_{A\otimes B}, t^\perp_{A\otimes B})$. The function $\mathcal{R}^\#_{A\otimes B} : \Sigma_{A\otimes B} \times \mathcal{T}_{A\otimes B} \to \Sigma_{A\otimes B}$ is defined as:*

$$\mathcal{R}^\#_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B)) = (\alpha_A(\sigma'^\perp_{A\otimes B}), \alpha_B(\sigma'^\perp_{A\otimes B})).$$

Practically speaking, however, one does not have access to the particular sequence of events occurring at $\perp$, i.e. to the ideal $\mathcal{R}^\#_\perp$ monitor and to precise concretization/abstraction functions for the containers. Nevertheless, it is possible to characterize sound approximations of composed monitors.

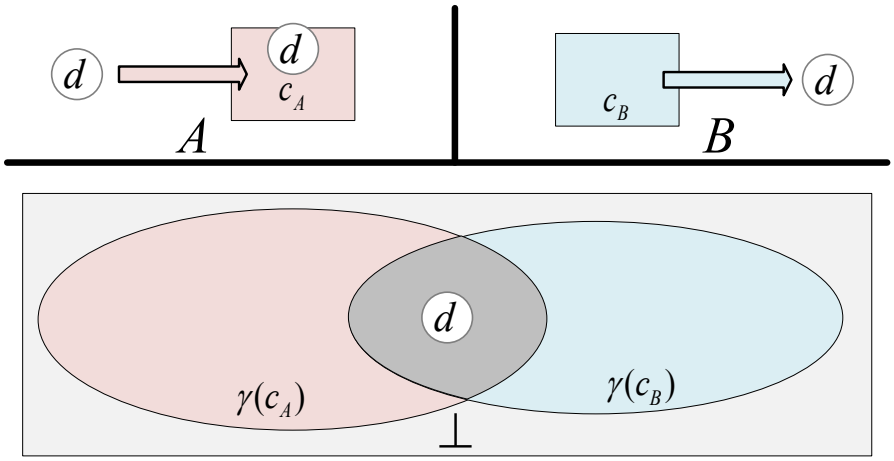


Figure 4.3.: Example of containers at different layers with overlapping concretizations. In this case, it is possible that data d, written to c_A by an event at layer A can be read out of c_B at layer B.

Definition 4.3 (Soundness of composing monitor)**.** *A monitor $\mathcal{R}_{A\otimes B}$ is sound w.r.t $\perp$, written $\mathcal{R}^{\#}_{A\otimes B} \vdash \mathcal{R}_{A\otimes B}$ if for all $\sigma_A, \sigma_B, t_A, t_B$ with*

$$\sigma' = \mathcal{R}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))$$

then the projections to A-layer containers $\sigma'|_A$ and B-layer containers $\sigma'|_B$ are sound with respect to $\mathcal{R}^{\#}_{\perp}(\gamma_A(\sigma_A) \bowtie_\sigma \gamma_B(\sigma_B), \gamma_A(t_A) \bowtie_t \gamma_B(t_B))$.

Note that, without any additional information about the relation between A and B, the only sound approximation for $\sigma' = \mathcal{R}_{A\otimes B}(\sigma, (t_A, t_B))$ is $\forall c \in \mathcal{C}_{A\otimes B} : \sigma'(c) = \mathcal{D}$, i.e. every container possibly contains any data. This is because there may be shared resources between layers that are unknown to the single monitors and are only visible at $\perp$: if there exist $c_A \in \mathcal{C}_A$ and $c_B \in \mathcal{C}_B$ such that $\gamma_A(c_A) \cap \gamma_B(c_B) \neq \emptyset$; in this case, it is possible that any data d may be transferred to c_A by some event $e_A \in \mathcal{S}_A$, and because of the non-empty intersection of the concretizations, d could be stored in $\gamma_B(c_B)$ too, as depicted in Figure 4.3.

Unless $d \in c_A$, this is a violation of the soundness of $A \otimes B$, because $c_A \in \mathcal{C}_A \subseteq \mathcal{C}_{A\otimes B}$ and while c_A does not contain d, its concretization does ($\sigma_{A\otimes B}(c_A) \not\supseteq \sigma_\perp(\gamma_A(c_A))$).

4.2.2. Simple Model

Definition 4.4 (Related Containers)**.** *Two containers c_A and c_B at different layers are called* related*, written $c_A \sim c_B$, if their concretizations overlap. Formally,*

$$\forall c_A \in \mathcal{C}_A, \forall c_B \in \mathcal{C}_B : c_A \sim c_B \iff \gamma_A(c_A) \cap \gamma_B(c_B) \neq \emptyset.$$

One approach to provide a sound estimation of the composed state $\sigma_{A\otimes B}$ is to maintain the content of related container "synchronized" across the layers, assuming such a relation between containers to be given.

Definition 4.5 (Simple Model)**.** *Leveraging the source and destination sets for events in the trace, a sound monitor for the composed system $\hat{\mathcal{R}}_{A\otimes B} : \Sigma_{A\otimes B} \times \mathcal{T}_{A\otimes B} \to \Sigma_{A\otimes B}$ can be defined as:*

$$\forall c \in \mathcal{C}_{A\otimes B} :$$
$$\hat{\mathcal{R}}_{A\otimes B}(\sigma, (t_A, t_B))(c) = \begin{cases} \sigma(c) \cup \bigcup_{c' \in \mathbb{S}_{A\otimes B}} \sigma(c') & \text{if } c \in \mathbb{D}_{A\otimes B} \vee \exists\, \tilde{c} \in \mathbb{D}_{A\otimes B} : c \sim \tilde{c} \\ \sigma(c) & \text{otherwise} \end{cases}$$

with $\mathbb{S}_{A\otimes B} = \mathbb{S}_{\mathcal{R}_A}(t_A) \cup \mathbb{S}_{\mathcal{R}_B}(t_B)$ and $\mathbb{D}_{A\otimes B} = \mathbb{D}_{\mathcal{R}_A}(t_A) \cup \mathbb{D}_{\mathcal{R}_B}(t_B)$.

Theorem 4.6 (Simple Model Soundness)**.** *Given a state $\sigma_\perp$, two states for two different layers of abstraction σ_A and σ_B sound w.r.t. $\sigma_\perp$ and the two respective transition functions $\mathcal{R}_A$ and $\mathcal{R}_B$ sound w.r.t. $\mathcal{R}^\#_\perp$, then $\hat{\mathcal{R}}_{A\otimes B}$ is sound.*

$$\sigma_\perp \vdash \sigma_A \wedge \sigma_\perp \vdash \sigma_B \wedge \mathcal{R}^\#_\perp \vdash \mathcal{R}_A \wedge \mathcal{R}^\#_\perp \vdash \mathcal{R}_B \implies \mathcal{R}^\#_{A\otimes B} \vdash \hat{\mathcal{R}}_{A\otimes B}$$

A detailed proof is given in Appendix B.1. The intuition behind the proof comes from the definitions of source and destination containers of events (see Section 3.4.2).

Let σ be the initial state of the system. The idea is that if the state σ' after the execution of trace $t \in \mathcal{T}_{A\otimes B}$ is not sound, then there must exists a data d stored in a $\perp$-container $c_\perp$ that is not present in container $c = \alpha_{A\otimes B}(c_\perp)$. This can only be the case if, either, d was already stored in $c_\perp$ in state σ or d has been transferred to $c_\perp$ by the execution of t.

The first case is not possible, because the soundness of σ implies that d was also stored in c and because $\hat{\mathcal{R}}_{A\otimes B}$ only appends data to containers, if d was stored in c in σ, it would still be there in σ'. The second case is not possible because, as the proof shows in detail, this would violate the definition of $\mathbb{S}_{A\otimes B}$ and $\mathbb{D}_{A\otimes B}$ for t. Therefore state σ' cannot be unsound, confirming the soundness of $\hat{\mathcal{R}}_{A\otimes B}$.

4.2.3. X_A Oracle

The definition of $\hat{\mathcal{R}}_{A\otimes B}$ relies on the notion of related containers, which in turn depends on the definition of γ. As discussed in Section 3.4, α and γ are usually not available *in practice* and thus cannot be used for the construction of $\hat{\mathcal{R}}_{A\otimes B}$. As also discussed before, without any information about the relation between A and B, the only sound approximation for $\hat{\mathcal{R}}_{A\otimes B}$ is the trivial monitor in which every container could possibly contain any data.

Although the complete definition of γ and α may not be available, it is often the case that, in some contexts, partial information about it is known by domain experts (e.g. the set of related containers). The goal of this section is to model this partial information in form of oracles and formalize refined data flow tracking models that, leveraging on properties of these oracles, provide more precise results. In the following the properties of these oracles will be formalized. Note that, operationally, such oracles will have to be instantiated by experts (see Section 4.4).

The core idea behind the oracles is that if more information about the relation between A and B is available, a more precise sound model can be constructed.

An instantiation of $\hat{\mathcal{R}}_{A\otimes B}$ requires the existence of an oracle $X_A : \mathcal{C}_{A\otimes B} \to \mathbb{P}(\mathcal{C}_{A\otimes B})$ that provides information about related containers: for each container $c \in \mathcal{C}_{A\otimes B}$, X_A returns the set of all the containers related to c at other layers, i.e.

Definition 4.7 (Oracle Assumption 1)**.**

$$\forall c \in \mathcal{C}_{A\otimes B} : X_A(c) = \{c' \in \mathcal{C}_{A\otimes B} \mid \exists l \in \mathcal{L} : c' \in \mathcal{C}_l \wedge c \notin \mathcal{C}_l \wedge c \sim c'\}.$$

Leveraging X_A, it is also possible to model the $sync$ operator, which will be useful in the following. Given a state of the system, the operator $sync : \Sigma_{A\otimes B} \to \Sigma_{A\otimes B}$ returns a new state in which all the data stored in each container have been propagated to all the related containers at other layers:

$$\forall c \in \mathcal{C}_{A\otimes B}, \sigma \in \Sigma_{A\otimes B} : sync(\sigma)(c) = \sigma(c) \cup \textstyle\bigcup_{c' \in X_A(c)} \sigma(c').$$

Because the sync operator only adds data to containers, it is easy to prove that if σ is a sound state (see Section 3.4.3), then $\sigma' = sync(\sigma)$ is also a sound state.

4.3. A Sound and Precise Cross-layer Monitor

In the model for usage control described in Chapter 3, events are assumed to be instantaneous, i.e. the time required to execute an event is considered to be negligible w.r.t to the length of a timestep. Although negligible at one layer, however, in a multi-layer context the duration of an event may cover several timesteps at the other layer. For instance, at the `EX` layer, the `SAVE()` event is considered as a single atomic event, although during the time required to execute a `SAVE()` event, many `WRITE()` system call events at layer `OS` may take place.

In a multilayer context, the notion of timestep must be granular enough to cover for the precision of both layers; for example, if the highest frequency at which `EX` can observe user events is every 100 milliseconds and layer `OS` observes system calls with the precision of a millisecond, then a model for the combined system must be able to observe events at least every millisecond.

For this reason, the duration of "long" events in the combined system (like `SAVE()`) is modeled with two atomic events that respectively represent the moment in which the *event starts* ($\texttt{SAVE}_S\texttt{()}$) and the moment in which the *event ends* ($\texttt{SAVE}_E\texttt{()}$).

Definition 4.8. *Let $t^S(e) : \mathcal{S} \to \mathbb{N}$ and $t^E(e) : \mathcal{S} \to \mathbb{N}$ be two functions that return, respectively, the time at which a certain event e starts and ends.*

Let $t_\dagger$ be a trace of events at layer $\dagger$. For any event $e_\dagger \in \mathcal{S}_\dagger$ it holds that $e_\dagger$ terminates only after starting ($t^S(e_\dagger) < t^E(e_\dagger)$) and for every event $e_\dagger$ observed, the single layer monitors report an event $e^S_\dagger$ at time $t^S(e_\dagger)$ to notify the beginning of $e_\dagger$ and an event $e^E_\dagger$ at time $t^E(e_\dagger)$ to notify its end.

Technically, this assumption may require a modification to the monitoring infrastructure, although in all the example instantiations mentioned in this dissertation, it has always been trivial to observe the duration of an event, and thus to replace the signaling of the event with two, one right before its execution and one right after.

For $\dagger \in \{A, B\}$, let $\mathcal{S}^-_\dagger \subseteq \mathcal{S}_\dagger \times \{S, E\}$ be the set of such *indexed* events that denote when events in $\mathcal{S}_\dagger$ start and end, and $\mathcal{T}^-_\dagger \subseteq seq(\mathcal{S}^-_\dagger)$ the set of respective system traces. Let $ser : seq(\mathcal{S}_\dagger) \to seq(\mathcal{S}^-_\dagger)$ be the operator that converts a trace of events $t_\dagger \in seq(\mathcal{S}_\dagger)$ into its indexed equivalent $t^-_\dagger \in seq(\mathcal{S}^-_\dagger)$ by replacing every event $e_\dagger \in t_\dagger$ with the sequence $\langle e^S_\dagger, e^E_\dagger \rangle$.

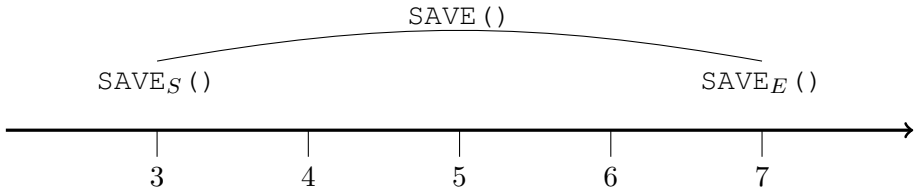


Figure 4.4.: Example of a SAVE () event lasting from timestep 3 to timestep 7 [137].

Lemma 4.9. *For each monitor $\mathcal{R}_\dagger$ ($\dagger \in \mathcal{L}$), there always exists a monitor $\mathcal{R}^-_\dagger : \Sigma_\dagger \times \mathcal{S}^-_\dagger \to \Sigma_\dagger$ such that $\forall \sigma_\dagger \in \Sigma_\dagger, \forall t_\dagger \in \mathcal{T}^-_\dagger : \mathcal{R}_\dagger(\sigma, t_\dagger) = \mathcal{R}^-_\dagger(\sigma, ser(t_\dagger))$.*

Proof. Given $\mathcal{R}_\dagger$, the monitor $\mathcal{R}^-_\dagger$, defined as

$$\mathcal{R}^-_\dagger(\sigma, (e_\dagger, i)) = \begin{cases} \sigma & \text{if } i = S \\ \mathcal{R}_\dagger(\sigma, e_\dagger) & \text{if } i = E \end{cases}$$

respects the property. □

It is thus possible to assume, without loss of generality, that every monitor for a layer $\dagger$ is defined over events in $\mathcal{S}^-_\dagger$; such monitor is denoted as $\mathcal{R}^-_\dagger$.

Definition 4.10 (Serializable trace)**.** *A trace $t = (t_A, t_B)$ is serializable if for every pair of events $e_A \in t_A, e_B \in t_B$, $t^S(e_A) \neq t^S(e_B)$ and $t^E(e_A) \neq t^E(e_B)$.*

Let $\mathcal{E}_{A\otimes B} = \mathcal{S}_A \cup \mathcal{S}_B$ and $\mathcal{S}^-_{A\otimes B} = \mathcal{S}_{A\otimes B} \times \{S, E\}$. Let $\mathcal{T}^-_{A\otimes B} \subseteq seq(\mathcal{S}^-_{A\otimes B})$ be the set of event traces in the composed system. For notation's simplicity's sake, let $\gamma_{A\otimes B}$ be the overloading of γ for $C_{A\otimes B}$, $\Sigma_{A\otimes B}$, events and traces of events in the composed system. If a trace $t = (t_A, t_B) \in seq(\mathcal{S}_A) \times seq(\mathcal{S}_B)$ is serializable, then it is possible to construct a trace $t^- \in \mathcal{T}^-_{A\otimes B}$ that is equivalent to t, in the sense that it is possible to reconstruct each one given the other. t^- is given by the events in $ser(t_A) \bowtie_t ser(t_B)$ sorted by timestamp. Let also assume $\mathcal{R}^{\#-}_{A\otimes B}$ to be the overloaded version of the ideal monitor for traces in $\mathcal{T}^-_{A\otimes B}$.

The monitor for the composed system $\dot{\mathcal{R}}_{A\otimes B}$ described at the end of this chapter (see Section 4.3.4) assumes the trace of input events $t = (t_A, t_B)$ to be serializable and provided in form of a sequence of events in $\mathcal{S}^-_{A\otimes B}$

$$\dot{\mathcal{R}}_{A\otimes B} : \Sigma_{A\otimes B} \times \mathcal{S}^-_{A\otimes B} \to \Sigma_{A\otimes B}$$

To simplify the notation, whenever a composed trace contains a certain event e_S directly followed by e_E, the pair of events is replaced by e. For instance, the trace of events $\langle \texttt{LOAD}_S\texttt{()}, \texttt{OPEN}_S\texttt{()}, \texttt{OPEN}_E\texttt{()}, \texttt{READ}_S\texttt{()}, \texttt{READ}_E\texttt{()}, \texttt{CLOSE}_S\texttt{()}, \texttt{CLOSE}_E\texttt{()}, \texttt{LOAD}_E\texttt{()} \rangle$ will be written $\langle \texttt{LOAD}_S\texttt{()}, \texttt{OPEN()}, \texttt{READ()}, \texttt{CLOSE()}, \texttt{LOAD}_E\texttt{()} \rangle$. For simplicity of notation, the monitor $\mathcal{R}^-_{\texttt{OS}}$ for the operating system used in the running example is assumed to be constructed from $\mathcal{R}_{\texttt{OS}}$ in the way described in the proof of Theorem 4.9. For this reason, in the remaining of this work start events at the OS layer will be ignored and only end events will be discussed.

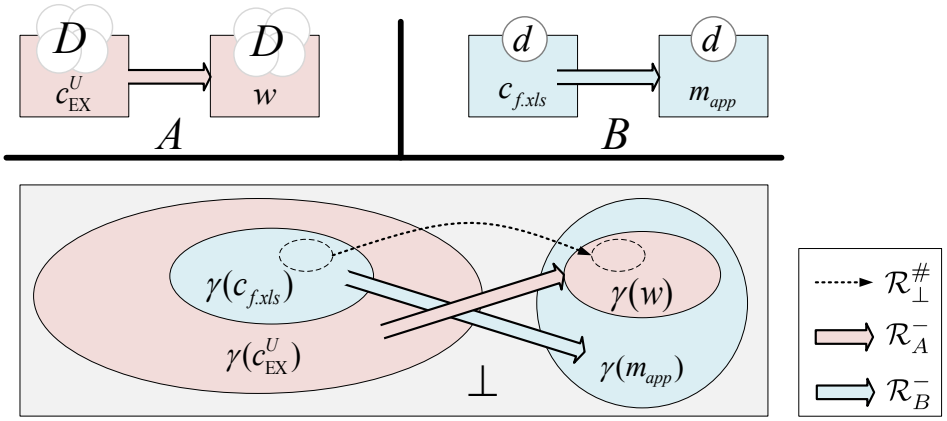


Figure 4.5.: Example of application loading a file, according to single layer monitors [109]. Dotted sets represent actual $\mathbb{S}_{\mathcal{R}^{\#}_{\perp}}$ and $\mathbb{D}_{\mathcal{R}^{\#}_{\perp}}$ sets.

4.3.1. Increasing Precision: Example

Consider the example in Section 4.1 of Excel loading file $f.xls$. The act of loading the file generates the trace

$$t = \langle \texttt{LOAD}_S\,(f.xls)\,, \texttt{OPEN}\,(f.xls, fd)\,, \texttt{READ}\,(fd)\,, \texttt{CLOSE}\,(fd)\,, \texttt{LOAD}_E\,(f.xls)\,\rangle$$

where the first and last events happen at layer EX and all other events at layer OS.

Because files are not properly modeled in EX, the source of the transfer in EX is given by $c^U_{\texttt{EX}}$ (see definition of c^U in Section 3.4). Because the file is unknown, it could possibly carry any data. This explains why $\forall \sigma_\dagger \in \Sigma_\dagger : \sigma_\dagger(c^U_\dagger) = \mathcal{D}$. The execution of $t|_{\texttt{EX}}$ induces then a flow of all data $\mathcal{D}$ from $c^U_{\texttt{EX}}$ to w, where w is an internal container of the application, e.g. a document. ($\mathcal{R}^-_A$ in Figure 4.5)

At the operating system layer, in contrast, the file has a proper abstraction. Let $c_{f.xls}$ be such a container and d the data stored in it. The execution of $t|_{\texttt{OS}}$ is then modeled in OS as a flow from $c_{f.xls}$ to the container m_{app} representing the whole memory of the application. ($\mathcal{R}^-_B$ in Figure 4.5)

If EX and OS were considered in isolation, the storage of w and m_{app} after the execution of t would be, respectively $\mathcal{D}$ and d. Using the model presented in Section 4.2.2 instead, both containers would contain $\mathcal{D}$, a sound but coarse approximation.

A better approximation can be provided by observing that $\gamma_{\texttt{OS}}(c_{f.xls}) \subseteq \gamma_{\texttt{EX}}(c^U_{\texttt{EX}})$ and $\gamma_{\texttt{EX}}(w) \subseteq \gamma_{\texttt{OS}}(m_{app})$, the latter because any internal object of the application is stored within its process memory. Assuming the application process has not accessed any other sensitive data, after the execution of t the content of all the $\perp$-containers in $\gamma_{\texttt{OS}}(m_{app})$, which include those in $\gamma_{\texttt{EX}}(w)$, is at most d, as reported by $\mathcal{R}^-_{\texttt{OS}}$ and because of its soundness. Therefore, no more data than d can be stored in $\gamma_{\texttt{EX}}(w)$.

Following this intuition, a more precise monitor for the combined system can be realized my modeling t as a flow from $\gamma_{\texttt{OS}}(f.xls)$ to $\gamma_{\texttt{EX}}(w)$. In the resulting state, both w and m_{app} contain d. Note that this result is more precise than $\mathcal{R}^-_{\texttt{EX}}$'s estimation.

A similar precision refinement would happen when Excel tries to save data d into a certain file. In this case, layer EX would provide a refinement for the operating system layer, which otherwise would transfer the whole content of m_{app} to the file, i.e. every data accessed by Excel until that moment of time (e.g. like data d and e in the File saving example in Section 4.1.2).

4.3.2. Event Behaviors

What the scenarios in the last section illustrate is that sometimes one layer has a more precise knowledge than the other about *the sources* of a certain event (e.g. the content of the file in the File loading example, the data to be saved in the File saving example), while the other layer has a finer-grained understanding of the *destination* of the transfer (e.g. the Excel specific container w in the File loading example, the target file in the File saving example). Let the term *cross actions* indicate those high-level operations, like '`EX` loading file $f.xls$' or '`EX` saving to file $f.xls$', that correspond to traces of events at both layers in which this refinement situation holds.

In the following, events generated by cross actions will be categorized according to their role in such "refinement": the terms `IN`, `OUT` and `INTRA` will be used to indicate the *behaviors* of events in cross actions.

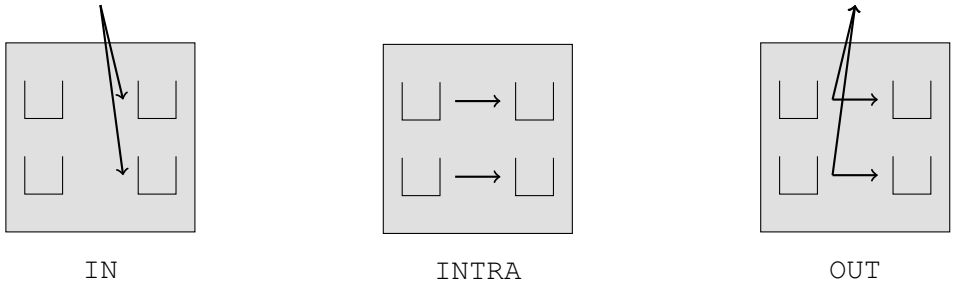


Figure 4.6.: Graphical representation of the behaviors of an event in cross-layer settings [137]. Squares indicate layer boundaries and arrows indicate data flows induced by the event's execution.

For instance, if a certain cross action generates two events $e_A \in \mathcal{S}_A^-$ and $e_B \in \mathcal{S}_B^-$ such that $\gamma_A(\mathbb{S}_{\mathcal{R}_A^-}(e_A)) \subseteq \gamma_B(\mathbb{S}_{\mathcal{R}_B^-}(e_B))$ and $\gamma_B(\mathbb{D}_{\mathcal{R}_B^-}(e_B)) \subseteq \gamma_A(\mathbb{D}_{\mathcal{R}_A^-}(e_A))$, e_A is an `OUT` event (or e_A *behaves* as an `OUT` event), and e_B is (or behaves as) an `IN` event. The general intuition is that `OUT` events at one layer provides data that is consumed by respective `IN` events at the other layer. This information can be used to refine the modeling as described in the previous example. If an event is neither `IN` nor `OUT` then it is an `INTRA` event.

In completely independent layers or when a layer is considered in isolation, every event is an `INTRA` event. In a multi-layer context an `INTRA` event at layer † propagates data within † according to $\mathcal{R}_\dagger^-$ and, in turn, to any other layer according to X_A.

Hence, in addition to the dependency between layers generated by related containers and discussed in Section 4.2.2, also a second class of cross-layer flows needs to be included into the model, i.e. the class of those cross-layer flows due to respective `IN` and `OUT` events.

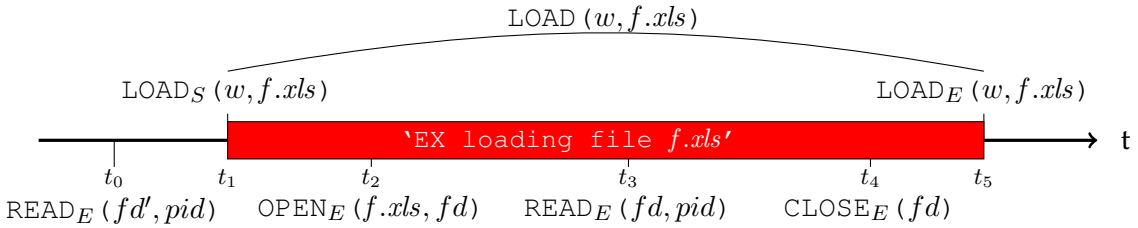


Figure 4.7.: Example of scope [137]. The cross action 'EX loading file $f.xls$' starts with event LOAD$_S$() and terminates with events LOAD$_E$().

Definition 4.11. *A* cross-layer flow *of data is generated by either:*

- *executing an event that transfers data to a container at one layer that is related with a container at the other layer, or*
- *a cross action generating a sequence of events at both layers that includes at least one* IN *event at one layer and at least one respective* OUT *event at the other layer.*

The intuition behind IN and OUT events is that, in spite of what the single layer monitors may estimate, the only data written to the destinations of a certain IN event is at most the same data read by the respective OUT events at the other layer. The next section formalizes this notion in form of an additional oracle.

4.3.3. X_B Oracle

In a multi-layer system, the behavior of a given event may differ in different contexts. For instance, in the example of Excel loading $f.xls$, a READ() event signaled by the operating system is related to a LOAD() event at the application layer, only if the process that invoked the system call is Excel and if the target file of the system call is $f.xls$. If the same file is read by another process, the behavior of the corresponding READ() event should be INTRA, because the system call would not be part of any cross action.

Similarly, if Excel is loading two files at the same time, then the trace of events contains two LOAD() events and at least two READ() events and a sound and precise modeling requires to match each LOAD() with the respective READ() events only.

To capture this distinction, the model assumes the existence of unique identifiers, called *scopes*, that relates IN and OUT events that pertain to the same cross action. Every cross action is associated to a distinct scope label, and only the events at both layers generated by the same cross action are associated to that same scope id. For instance, consider the example in Figure 4.7: while the first READ() system call does not correspond to any cross action, and as such it is modeled as an INTRA event, the second one happens in the context of Excel loading $f.xls$. For this reason, it is modeled as an OUT event with respect to the scope 'EX loading file $f.xls$'.

The need for this distinction motivates the definition of the second oracle X_B of type

$$X_B : \mathcal{S}^{-}_{A \otimes B} \times \Sigma_{A \otimes B} \rightarrow \{\texttt{IN}, \texttt{OUT}, \texttt{INTRA}\} \times SCOPE$$

where $SCOPE$ is the set of cross action's labels, like 'EX loading file $f.xls$'. X_B maps each event to its respective behavior in the context of a specific cross action. In the

loading example, $X_B(\texttt{READ}(fd, pid), \sigma) = (\texttt{OUT}, \texttt{`EX loading file } f.xls\texttt{'})$, where σ is the state after executing the trace until time 2 included.

Note that the state of the system in terms of storage is usually not enough to distinguish the behavior of a certain event. Consider again the example in Figure 4.7: In terms of storage, the state of the system at time t_0, i.e. when the first `READ()` takes place, is the same as the state at time t_2, when the second `READ()` is observed. In addition to the concrete parameters of the system calls, the difference is in the fact that the second `READ()` system call takes place after the loading of $f.xls$ started (i.e. after $\texttt{LOAD}_S\texttt{()}$) and before it ended.

In other words, the behavior of an event depends on the current state of the system, the value of its parameters and *the sequence of events that took place so far*. For this reason, every trace of events in the combined model is also stored as part of the state resulting from its execution.

Intermediate Containers

It is possible that multiple `IN` (`OUT`) events correspond to the same `OUT` (`IN`) event, e.g. one `LOAD()` event may correspond to multiple `READ()` system calls. Additionally, in serialized traces, respective `IN` and `OUT` events take place in different moments in time. For these reasons, it is necessary to aggregate and to store the content of the data being transferred by the `OUT` events in a way that it is usable by the future corresponding `IN` events.

A more precise definition for a composed system also models the existence of a container c_{sc} for each scope $sc \in SCOPE$. Every `OUT` event of scope sc will write into c_{sc} and every `IN` event of scope sc will read from c_{sc}. Container $c_{sc} \in \mathcal{C}_{sc}$ is called the *intermediate* container of the cross layer flow sc. Storage information for the intermediate containers must also be maintained as part of the system state in form of storage function $s_{sc} : \mathcal{C}_{sc} \to \mathbb{P}(\mathcal{D})$.

Let c_s be a source of an `OUT` event and c_d a destination of the respective `IN` event. The flow from c_s to c_d is modeled in two steps: first as a flow from c_s to the intermediate container c_{sc} and then as a flow from c_{sc} to c_d (see Figure 4.8 for an example).

For this reason, this work considers only serialized traces where `IN` events take place after the respective `OUT` events[2]. This assumption is not restrictive in practice and always held in concrete instantiations of the model.

In summary, the relation between two given layers A and B can be encoded by using the oracles

$$\begin{aligned} X_A: &\quad C_{A\otimes B} \to 2^{C_{A\otimes B}} \\ X_B: &\quad \Sigma_{A\otimes B} \times \mathcal{S}^-_{A\otimes B} \to \{\texttt{IN}, \texttt{OUT}, \texttt{INTRA}\} \times SCOPE \end{aligned}$$

Let t_e denote the subtrace of events in trace t from the beginning until event e included, and let σ^e be a short notation for the state reached by the ideal monitor $\mathcal{R}^{\#-}_{A\otimes B}$ after executing t_e from the initial state, i.e. $\sigma^e = \mathcal{R}^{\#-}_{A\otimes B}(\sigma_i, t_e)$. The oracles, by definition, guarantee the following property:

[2] The case of `IN` events taking place before the respective `OUT` events can be modeled with additional assumptions, e.g. using a placeholder to overapproximate the content after execution of `IN` events and replacing it with the precise value after execution of the corresponding `OUT` event, but is not discussed in this work for simplicity's sake.

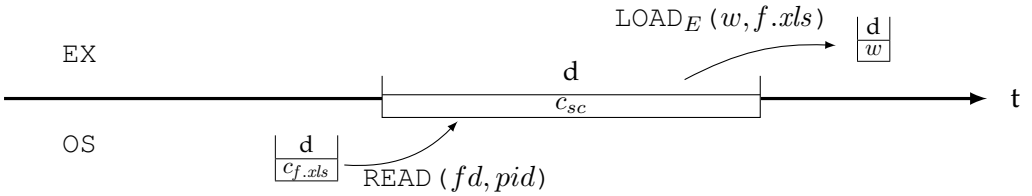


Figure 4.8.: Graphic representation of intermediate container for scope sc = 'EX loading file $f.xls$', where c_{sc} stores data d read by the (OUT) READ (fd, pid) system call at layer OS until it is read by the (IN) LOAD$_E$ $(w, f.xls)$ event in EX [137].

Definition 4.12 (Oracle Assumption 2). *Let $t \in \mathcal{T}^-_{A\otimes B}$ be a trace of events terminating with the IN event e^I and let $E^O \subseteq \mathcal{S}^-_{A\otimes B}$ be the set of respective OUT events in t. After executing t in an ideal monitoring, the destination containers of e^I contain at most the content of the sources of all the events in e^O at the time of their execution.*

$$\left(\begin{array}{c} X_B(\sigma^{e^I}, e^I) = (IN, sc) \wedge \\ \forall e \in E^O : X_B(\sigma^e, e) = (OUT, sc) \end{array}\right) \implies \sigma^{e^I}(\mathbb{D}_{\mathcal{R}^\#}(e^I)) \subseteq \bigcup_{e\in E^O} \sigma^e(\mathbb{S}_{\mathcal{R}^\#}(e))$$

where $\mathcal{R}^\#$ stands for $\mathcal{R}^{\#-}_{A\otimes B}$.

The intuition behind this property is that if the oracle X_B states that a certain event e is an IN event in a trace, then the execution of e will transfer to e's destination containers at most the data stored in the sources of the respective OUT events in the past trace. This intuition is the key behind the refined precision offered by $\dot{\mathcal{R}}_{A\otimes B}$ in comparison to $\hat{\mathcal{R}}_{A\otimes B}$.

4.3.4. Refined Model and Algorithm

Given the models for A and B and these two oracles, the model $A \otimes B$ for the composed system is specified as follows:

$$(\mathcal{D}, \mathcal{S}^-_{A\otimes B}, \mathcal{C}_{A\otimes B}, \Sigma_{A\otimes B}, \sigma_i, \dot{\mathcal{R}}_{A\otimes B})$$

The set of containers $\mathcal{C}_{A\otimes B}$ is given by $\mathcal{C}_A \cup \mathcal{C}_B \cup \mathcal{C}_{sc}$, with $\mathcal{C}_{sc}$ being the set of intermediate containers. Remember that containers in $\mathcal{C}_{sc}$ represent no real container in the system, i.e. $\forall c \in \mathcal{C}_{sc} : \gamma_{A\otimes B}(c) = \emptyset$. A state of the system $\sigma_{A\otimes B} \in \Sigma_{A\otimes B}$ corresponds to the state of the two layers A and B, the storage function for intermediate containers $s_{sc} : \mathcal{C}_{sc} \to \mathbb{P}(\mathcal{D})$ and the trace of past events $tr \in \mathcal{T}^-_{A\otimes B}$.

$$\begin{aligned} &\mathcal{C}_{A\otimes B} = \mathcal{C}_A \cup \mathcal{C}_B \cup \mathcal{C}_{sc} \\ &\Sigma_{A\otimes B} \subseteq \Sigma_A \times \Sigma_B \times (\mathcal{C}_{sc} \to \mathbb{P}(\mathcal{D})) \times \mathcal{T}^-_{A\otimes B} \end{aligned}$$

The main result of this thesis is to show that a composition algorithm for cross-layer data flow tracking, based on the aforementioned oracles, is sound w.r.t. to an ideal monitor at $\perp$, and thus to weak-secrecy.

Given two sound instantiations of the generic model for two layers A and B and the two oracles defined above, a sound and precise modeling of the data flows within and across these two layers is captured by the transition relation $\dot{\mathcal{R}}_{A\otimes B}$ defined in Algorithm 1. $a||b$ denotes the concatenation of traces a and b. Note that the formalization considers the generic case where σ_A and σ_B are in turn composed by multiple layers, i.e. they may be instances of the cross-layer model themselves. Thus $\sigma = (\sigma_A, \sigma_B, s_{sc}, tr) = (s_A \cup s_B \cup s_{sc}, tr_A \bowtie_t tr_B)$.

ALGORITHM 1: $\dot{\mathcal{R}}_{A\otimes B}((\sigma_A, \sigma_B, s_{sc}, tr), e)$

$s_{sc_{RET}} \longleftarrow s_{sc}$; $\sigma_{A_{RET}} \longleftarrow \sigma_A$; $\sigma_{B_{RET}} \longleftarrow \sigma_B$;
$(beh, sc) \longleftarrow X_B((\sigma_A, \sigma_B, s_{sc}, tr), e)$;
switch *beh* **do**
  case `INTRA`
    if $e \in \mathcal{S}_A^-$ **then**
      $\sigma_{A_{RET}} \longleftarrow \mathcal{R}_A^-(\sigma_A, e)$;
    else
      $\sigma_{B_{RET}} \longleftarrow \mathcal{R}_B^-(\sigma_B, e)$;
  case `IN`
    if $e \in \mathcal{S}_A^-$ **then**
      $\sigma_{A_{RET}} \longleftarrow (s_A[t \leftarrow s_A(t) \cup s_{sc}(c_{sc})]_{t \in \mathbb{D}_{\mathcal{R}_A^-}(e)})$;
    else
      $\sigma_{B_{RET}} \longleftarrow (s_B[t \leftarrow s_B(t) \cup s_{sc}(c_{sc})]_{t \in \mathbb{D}_{\mathcal{R}_A^-}(e)})$;
  case `OUT`
    if $e \in \mathcal{S}_A^-$ **then**
      $s_{sc_{RET}} \longleftarrow s_{sc}[c_{sc} \leftarrow s_A(t)]_{t \in \mathbb{S}_{\mathcal{R}_A^-}(e)}$;
      $\sigma_{A_{RET}} \longleftarrow \mathcal{R}_A^-(\sigma_A, e)$;
    else
      $s_{sc_{RET}} \longleftarrow s_{sc}[c_{sc} \leftarrow s_B(t)]_{t \in \mathbb{S}_{\mathcal{R}_A^-}(e)}$;
      $\sigma_{B_{RET}} \longleftarrow \mathcal{R}_B^-(\sigma_B, e)$;
endsw
return $sync(\sigma_{A_{RET}}, \sigma_{B_{RET}}, s_{sc_{RET}}, tr||\langle e \rangle)$

Theorem 4.13. *Given two oracles X_A and X_B, correct according to oracle assumptions 1 and 2, two monitors for two layers of abstraction $\mathcal{R}_A^-$, $\mathcal{R}_B^-$, an initial state $\sigma_{A\otimes B} = (\sigma_A, \sigma_B, \langle\rangle)$ and a serializable trace of events $t \in \mathcal{T}_{A\otimes B}^-$, if σ_A and σ_B are sound w.r.t. $\sigma_\perp$ and $\mathcal{R}_A^-$ and $\mathcal{R}_B^-$ are sound w.r.t $\mathcal{R}_\perp$, then $\dot{\mathcal{R}}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))$ is sound w.r.t. $\mathcal{R}_{A\otimes B}^\#$.*

$$\begin{pmatrix} \sigma_\perp \vdash \sigma_A & \wedge \\ \sigma_\perp \vdash \sigma_B & \wedge \\ \mathcal{R}_\perp \vdash \mathcal{R}_A^- & \wedge \\ \mathcal{R}_\perp \vdash \mathcal{R}_B^- & \end{pmatrix} \implies \mathcal{R}_{A\otimes B}^\# \vdash \dot{\mathcal{R}}_{A\otimes B}.$$

A detailed proof is provided in Appendix B.2. The intuition is that, for INTRA events, $\dot{\mathcal{R}}_{A\otimes B}$ behaves similarly to $\hat{\mathcal{R}}_{A\otimes B}$, and therefore it is sound, and for OUT events related to a scope sc, the content of the sources is also stored in a container c_{sc}, from where it can be "read" by the corresponding IN events and transferred to their destinations. The soundness then comes from Definition 4.12.

Note that the assumption on the serializability of the input trace for $\dot{\mathcal{R}}_{A\otimes B}$ is not restrictive, because any trace of events $t_{A\otimes B} = (t_A, t_B)$ in $A \otimes B$ can be seen as the longest possible concatenation of subtraces $t_i = (t_{iA}, t_{iB})$, such that

- any event starting in t_i also terminates within t_i, and vice versa ($\forall e \in \mathcal{S}_A \cup \mathcal{S}_B : (e, S) \in t_i \iff (e, E) \in t_i$); and
- the concatenation of all the t_{iA}s corresponds to t_A and the concatenation of all the t_{iB}s corresponds to t_B ($(t_{1A}||t_{2A}||\ldots||t_{nA}) = t_A \wedge (t_{1B}||t_{2B}||\ldots||t_{nB}) = t_B$).

Then, for each t_i,

$$\mathcal{R}_{A\otimes B}(\sigma, t_i) = \begin{cases} \dot{\mathcal{R}}_{A\otimes B}(\sigma, t_i) & \text{if } t_i \text{ is serializable} \\ \hat{\mathcal{R}}_{A\otimes B}(\sigma, t_i) & \text{otherwise} \end{cases}$$

is a sound monitor that is no less precise than $\hat{\mathcal{R}}_{A\otimes B}(\sigma, t)$ and does not require t to be serializable.

4.4. Use Case (Multi-layer)

This section describes in detail a simplified concrete instantiation of the combined model for the two layers used as running example (EX and OS, see Section 4.1).

4.4.1. Instantiation of X_A

When combining EX with OS, the behavior of SAVE() and LOAD(), as well as that of WRITE() and READ(), should reflect the fact that data is being sent/read from across boundaries.

In terms of the first oracle, X_A a concrete implementation needs to model the fact that every container in EX is related to the container that represents the memory of the Excel process at OS layer. For the sake of simplicity, in this example this connection is assumed to be statically defined as

$$X_A = \{(c_{\text{EX}}, c_{\text{OS}}) | c_{\text{EX}} \in \mathcal{C}_{\text{EX}} \wedge c_{\text{OS}} = m_{app}\}$$

where m_{app} is the container representing the memory of the Excel process.

A real implementation, however, would also take into account that the Excel process may be killed and then restarted, resulting in a different m_{app} container every time. In that case, the definition of the X_A oracle would need to be updated accordingly, e.g. in correspondence of the system calls EXIT() that kill the Excel instance and CREATE_PROCESS() that start the new instance.

4.4.2. Instantiation of X_B

The definition of function $X_B()$ is presented in Algorithm 2. As discussed in Section 4.3.3, the different behavior of the events depend on the current state of the system and on the past trace of events. For instance, consider the code for the $\texttt{WRITE}_E\texttt{()}$ events, between line 2 and line 11 of Algorithm 2.

In order to decide if the behavior of the event is `INTRA` or `IN`, the oracle checks whether at the current moment in time there is any saving cross action of workbook w to file f being performed. This is done in line 5 by checking if in the past trace any $\texttt{SAVE}(w, f)$ event started ($\texttt{SAVE}_S(w, f) \in tr$) and not yet finished ($\texttt{SAVE}_E(w, f) \notin tr$).

ALGORITHM 2: $X_B((\sigma_A, \sigma_B, s_{sc}, tr), e)$

```
1  switch e do
2      case WRITE_E (fd, pid)
3          for each file f in OS do
4              for each workbook w in EX do
5                  if (SAVE_S (w, f) ∈ tr) ∧ (SAVE_E (w, f) ∉ tr) then
6                      sc ⟵ 'EX saving workbook w to file f';
7                      if (pid = ⟨pid of Excel⟩ ) ∧ (σ.f(f) = σ.f(fd, pid)) then
8                          return (IN, sc)
9                      end
10                 end
11         end
12     case READ_E (fd, pid)
13         for each file f in OS do
14             for each workbook w in EX do
15                 if (LOAD_S (w, f) ∈ tr) ∧ (LOAD_E (w, f) ∉ tr) then
16                     sc ⟵ 'EX loading workbook w from file f';
17                     if (pid = ⟨pid of Excel⟩ ) ∧ (σ.f(f) = σ.f(fd, pid)) then
18                         return (OUT, sc)
19                     end
20                 end
21         end
22     case SAVE_S (w, f.xls)
23         return (OUT, 'EX saving workbook w to file f');
24     case LOAD_E (w, f.xls)
25         return (IN, 'EX loading workbook w from file f');
26     otherwise
27         return (INTRA, ∅)
28     end
29 endsw
```

The dual check is performed for the `READ()` system calls, while for any $\texttt{SAVE}_S\texttt{()}$ and $\texttt{LOAD}_E\texttt{()}$ events, the behavior does not require any particular check on the past trace, because each instance of these events is relative to a different scope. This is a consequence

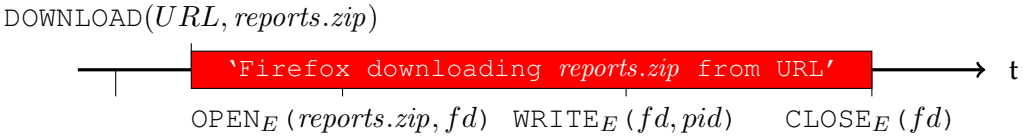


Figure 4.9.: Example of cross action initiated by an event at one layer (DOWNLOAD(P, *reports.zip*)) and terminated by an event at the other layer (CLOSE (fd)). Adapted from [137].

of the fact that in the running example the LOAD () event at layer EX is blocking and any corresponding event at layer OS takes place in-between the beginning (LOAD$_S$ ()) and the end (LOAD$_E$ ()) of such event and the same holds for SAVE (): every cross action that determines a cross-layer flow always starts and ends together with the respective EX event.

In general, however, this is not necessarily the case. The cross action may be initiated and terminated by events at any (and not necessarily the same) layer. For instance, consider the additional example of OS and a monitor for a browser application, like Mozilla Firefox, and the scenario of a user downloading file reports.zip from URL.

The Firefox event DOWNLOAD(URL, *reports.zip*) is not blocking, but returns immediately while the download proceeds in background until file *reports.zip* is completely saved on the local machine. This means that the end event DOWNLOAD$_E$(URL, *reports.zip*) will probably appear in the trace before the last WRITE () system call associated to the 'Firefox downloading *reports.zip* from URL' cross action.

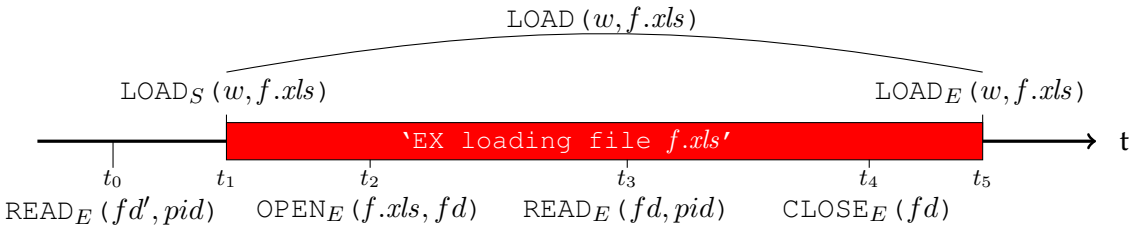
Therefore, in order to decide the IN behavior of the WRITE$_E$ (fd, pid) system calls, the X_B definition will check the past trace for the presence of the event that started the download (DOWNLOAD$_S$(URL, *reports.zip*)) and for the lack of the respective CLOSE$_E$ (fd, pid), where fd is the file descriptor pointing to file *reports.zip* and pid is the process id of Firefox.

4.4.3. Step-by-step Example

Figure 4.10 shows a Step-by-step execution of the File loading example discussed throughout this whole chapter. For shortness of notation, let $sc =$ 'EX loading file $f.xls$' and $pid = \langle$process id of Excel$\rangle$; for the same reason, the past trace tr is also not reported.

Assuming an initial state where a file descriptor fd' for a certain file $f.xls'$ have been already created, the first READ$_E$ (), at time t_0, behaves according to its intra-layer semantics and transfers the content of file $f.xls'$ to m_{app}, the container that represents the memory of the Excel process. After LOAD$_S$ () initiates the cross action $sc =$ 'EX loading file $f.xls$' at time t_1, the OPEN$_E$ () system call at time t_2 behaves according to its intra-layer semantics and creates a new file pointer fd to file $f.xls$. The READ$_E$ () system call at time t_3 behaves according to its OUT semantics and transfers the data in file $f.xls$ to $m_{app}$and to the intermediate container c_{sc}. CLOSE () system call at time t_4 closes the file descriptor fd and finally LOAD$_E$ () at time t_5 concludes the cross action transferring the content of c_{sc} to the container w (and, via X_A, also to m_{app}, although it already contained it).

The final state of the system reflects the fact that the content of file $f.xls$ at the operating system layer has been loaded into the Excel container w. Note the difference between the execution of the READ () system calls at time t_0 and at time t_3.



t	e	$X_B(\sigma, e)$	$\sigma' = R(\sigma, e) = \{\{s, l, f\}, tr\}$		
			s	l	f
		Initial state:	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$
t_0	READ$_E$ (fd', pid)	(INTRA, $\emptyset$)	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$ (m_{app}, e)		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$
t_1	LOAD$_S$ $(w, f.xls)$	(INTRA, $\emptyset$)	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$ (m_{app}, e)		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$
t_2	OPEN$_E$ $(f.xls, fd)$	(INTRA, $\emptyset$)	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$ (m_{app}, e)		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$ $fd \to c_{f.xls}$
t_3	READ$_E$ (fd, pid)	(OUT, sc)	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$ $(m_{app}, \{d, e\})$ (c_{sc}, d)		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$ $fd \to c_{f.xls}$
t_4	CLOSE$_E$ (fd)	(INTRA, $\emptyset$)	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$ $(m_{app}, \{d, e\})$ (c_{sc}, d)		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$
t_5	LOAD$_E$ $(w, f.xls)$	(IN, sc)	$(c_{f.xls'}, e)$ $(c_{f.xls}, d)$ $(m_{app}, \{d, e\})$ (c_{sc}, d) (w, d)		$f.xls' \to c_{f.xls'}$ $f.xls \to c_{f.xls}$ $fd' \to c_{f.xls'}$

Figure 4.10.: Step-by-step example of cross-layer data flow [137]. For shortness, sc denotes scope 'EX loading file $f.xls$', $pid = \langle$process id of Excel$\rangle$ and m_{app} the container representing the memory of the Excel process. Past trace of events tr and start events at OS layer are not reported.

4.5. Conclusions

This chapter investigated the problem of combining different instances of the generic model for data flow tracking described in Chapter 3. It showed that given a system composed by two layers of abstraction A and B, a *sound* tracking of flows of data from containers in A to containers in B and vice versa requires the specification of the relationship between A and B; the pure knowledge of the models for A and B does not suffice.

The model for two layers generalizes by recursion to the case for n layers: as mentioned at the beginning of this chapter, if the system is composed by $n > 2$ layers, then the same approach described in these pages can be applied recursively considering the $n-$th layer as A and the composition of the other $n - 1$ layers as B. Note that the $\bowtie_t$ operator is associative, i.e. given three traces t_A, t_B and t_C at three different layers of abstraction, $(\gamma(t_A) \bowtie_t \gamma(t_B)) \bowtie_t \gamma(t_C) = \gamma(t_A) \bowtie_t (\gamma(t_B) \bowtie_t \gamma(t_C))$. Considering that all the different monitored layers represent abstractions of the same real system, and thus concretize to the same $\perp$-layer, the associativity of the $\bowtie_t$ operator imposes no constraints about the order in which the layers are combined or to the general applicability of the approach. Note that the definition of $\dot{\mathcal{R}}_{A\otimes B}$ already supports the possibility that A and B are, in turn, instances of the cross layer model (see Section 4.3.4).

The only aspect that is affected by the combining order is the notion of `INTRA`, `IN` and `OUT`, which depends on the layer boundaries; for instance, the same event at layer A may be an `IN` event when combining A and B, and an `INTRA` event at layer AB when merging C with AB, where AB is the result of the combination of A and B. For this reason, the information provided by the oracles depends on the pair of layers considered, and thus, for more than 2 layers, on the order of the combination.

This chapter also showed that with additional information about the relation between A and B, it is possible to obtain information about data dissemination that is more *precise* than what is offered by the data flow tracking at the single layers considered in isolation. While in each specific instance, the problem of tracking flows of data across layers of abstractions can be easily solved with ad-hoc solutions [138, 41, 122], the model presented in this chapter is the first approach that aims at generalizing the process, separating aspects that are common to every instantiation (e.g. the definition of $\dot{\mathcal{R}}_{A\otimes B}$) from those specific of each single domain (e.g. the definition of the oracles).

These results address the fundamental research questions presented at the beginning of this chapter. In particular, Section 4.2 formalizes the relation between data representations and events at different layers of abstraction, Definition 4.11 captures the notion of flows of data across different layers of abstraction, and Section 4.2.1 provides a formal definition of soundness for cross-layer data flow tracking, based on the single layers' soundness defined in Section 3.4.

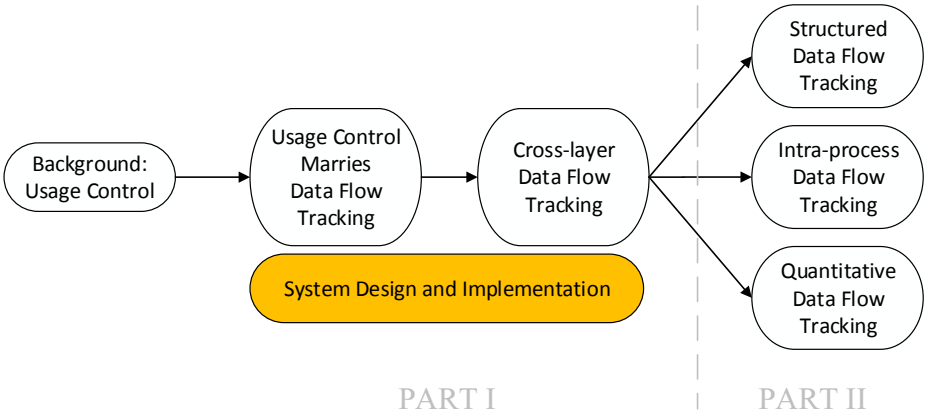
It is worth noting that this work describes only three different behaviors for events (`INTRA`,`IN` and `OUT`); this is a simplistic assumption, because semantics of events that indeed behave differently in a multi-layer context do not always fall into one of these three categories. For instance, in a multi-layer context some events may modify intra-layer alias relationships at another layer or some `OUT` events may overwrite the content of the intermediate container whilst other `OUT` events may simply append to it. The model described in these pages takes safe overapproximations for these cases, but depending on how precise the general model should be, one can specify many different additional behaviors.

Each behavior must then be encoded in $\dot{\mathcal{R}}_{A\otimes B}$, in order to define its semantics, and the definition of the X_B oracle needs to be accordingly updated.

Taking this approach to the extreme case where a different behavior is specified for each event in the system, the resulting model would be equivalent to a manual specification of the transition relation $\mathcal{R}$ for the complete system; this, in terms of effort, is equivalent to (or even worse than) an ad-hoc solution. Therefore, such a solution would obviously nullify the benefit of a generic model. In concrete instantiations of this model (see Section 9.1), the minimal set of cross-layer behaviors described in this chapter has always been enough to capture all the relevant cross-layer flows with an adequate level of precision.

5. System Design and Implementation

This chapter describes a generic architecture to instantiate the concepts described in the previous chapters. Despite some similarities with the XACML [143] system model, this architecture is part of the author's contribution and has been published in [138] and [100], both co-authored by the author of this dissertation.

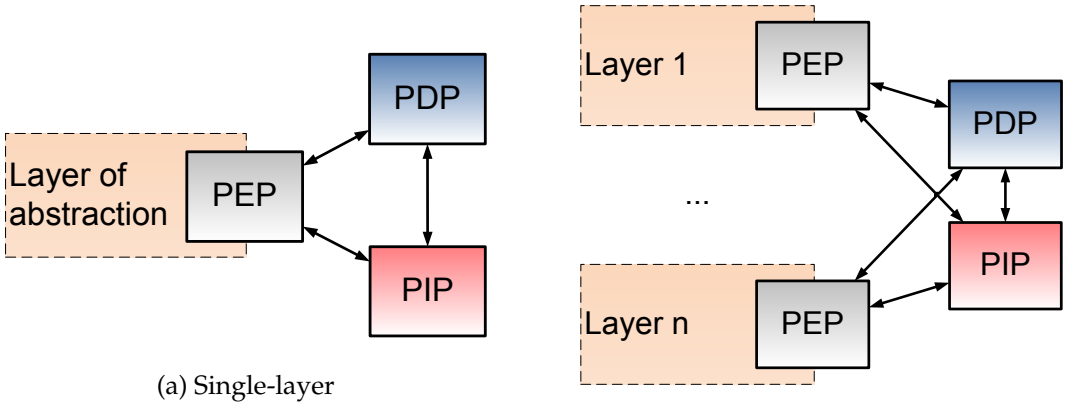


5.1. Architecture

The concepts described in the previous chapters can be operationalized in a layer-agnostic generic architecture. Such architecture is formed by three main components, depicted in Figure 5.1:

- a *Policy Enforcement Point* or PEP, which is in charge of intercepting system events and notifying them to the decision point;
- a *Policy Decision Point* or PDP, which receives events from the PEP and checks whether their execution complies with the security policy;
- a *Policy Information Point* or PIP, that keeps track of the dissemination of data throughout the system and supports the PDP in the decision process.

The following sections elaborate on the internal behavior and interaction between these components. Note that, as mentioned in Section 1.3, the negotiation phase with the data provider, the deployment of the security policies to be enforced, nor the subsequent creation of sensitive data in the initial representation of containers are discussed here: these topics are secondary to the goal of this work and are discussed in related work from the literature [138, 98, 99, 81].



(a) Single-layer

(b) Multiple Layers

Figure 5.1.: Interplay of the components in different settings

5.1.1. Policy Enforcement Point

The role of the PEP is to implement the ILP enforcement mechanisms described in Section 2.3.2. PEPs intercept intended and actual events, signal them to the PDP and, according to the response, allow, inhibit or modify them. Additionally, the PEP is also responsible of executing compensative actions, like those in the "action" part of an executor ILP (see Section 2.3).

Because of the nature of its role, the implementation of any policy enforcement point is bound to a specific layer of abstraction. In different contexts, the architecture presented in this work has been used with implementations of PEPs for different layers of abstraction, including OpenBSD [73], Android [55], Windows [164], Chromium [163], Thunderbird [105], Firefox in conjunction with social networks [100, 111], smart metering infrastructures [97, 59], an enterprise service bus [126], and camera surveillance systems [26].

5.1.2. Policy Decision Point

The role of the Policy Decision Point is to check whether the events signaled by the PEP violate the policies, or, for intended events, would violate the policies if executed. More precisely, the PDP checks if the execution of an event would make the condition of any ECA rule evaluate to true, i.e. if the event triggers the mechanism.

The PDP component represents, conceptually, the core of the whole architecture. While the implementations of the PDP mentioned in this dissertation are based on one specific algorithm [75], any runtime verification algorithm can also be used [102]. In contrast to the PEP, the generic nature of the PDP makes it possible to reuse the same implementation for instantiations of the model for different system layers: only the binding of events in the system to events specified in the policies has to be performed. Nevertheless, different versions of the PDP implementation, possibly written in different languages, have been used in different works (see Section 9.1), often with different features and capabilities. The implementations used for the evaluation of the models in the second part of this dissertation are no exception. For this reason, precision and performance evaluation details of each

extension implementation are described separately in the respective evaluation sections.

The languages Φ and Φ^- described in the previous chapters are based on temporal logic; the large amount of already available work on runtime monitoring [102, 24, 23, 25] can thus be exploited to synthesize efficient monitors both for specification-level and (the condition part of) implementation-level policies. The implementation of the evolution of the data state, necessary to implement Φ_s, is described in the next section.

5.1.3. Policy Information Point

In order to take a decision, the PDP may require additional information concerning the dissemination of data among the different representations (e.g., to decide about a state-based formula or a data usage event). For this reason the PDP queries the Policy Information Point. The PIP represents a (layer-specific) implementation of the data flow tracking model presented in Chapter 3. To properly maintain the correct status of the system, the PIP updates its data state $\sigma \in \Sigma$ according to each event it receives from the PEP.

At the implementation level, different solutions can be used to store the data state, ranging from databases to files, and from dynamic linked lists to arrays, depending on the different needs of the specific layer of abstraction. Conceptually, however, they all provide the same functionality. The implementation of the state transition relation, necessary to implement Φ_s, is straightforward: at each moment in time, any intercepted event is notified by the PEP to the PIP, which updates the data state according to the transition function ϱ. This simple implementation yields a state machine that computes the data state at every given moment in time. If a data usage event or a state-based operator is used in the specification of a policy (and thus in the synthesized monitor), the PDP can consult such state machine to retrieve all the containers that contain the respective data item, and evaluate the policy with respect to *all these containers*.

5.1.4. Interplay

The interplay of PEP, PDP, and PIP is shown in Figure 5.2. Whenever the PDP checks an actual (containerUsage) event e against a data usage event u in a policy, the PIP is consulted to check if the data item referred to by u is contained in the container referred to by e.

Single-layer

In the single-layer case, the PEP intercepts desired and actual events of a specific system layer and forwards them to the PDP for evaluation. Based on the PDP's decision the PEP either allows, inhibits, or modifies the signaled event, and potentially generates additional events if a matching policy demands so (Executors, see Section 2.3.2).

To conduct this evaluation, the PDP tries to match the events received from the PEP against deployed ILPs (events refinement, see Section 3.2). To evaluate the corresponding policies, in particular the condition part of the ECA rules, (see Section 2.3) the PDP uses a runtime verification algorithm. In some cases, e.g. if the trigger part of an ECA rule concerns a data-usage event (see Section 3.2), or the condition part contains a state-based formula (see Section 3.2.2), the PDP needs additional information for the policy evaluation.

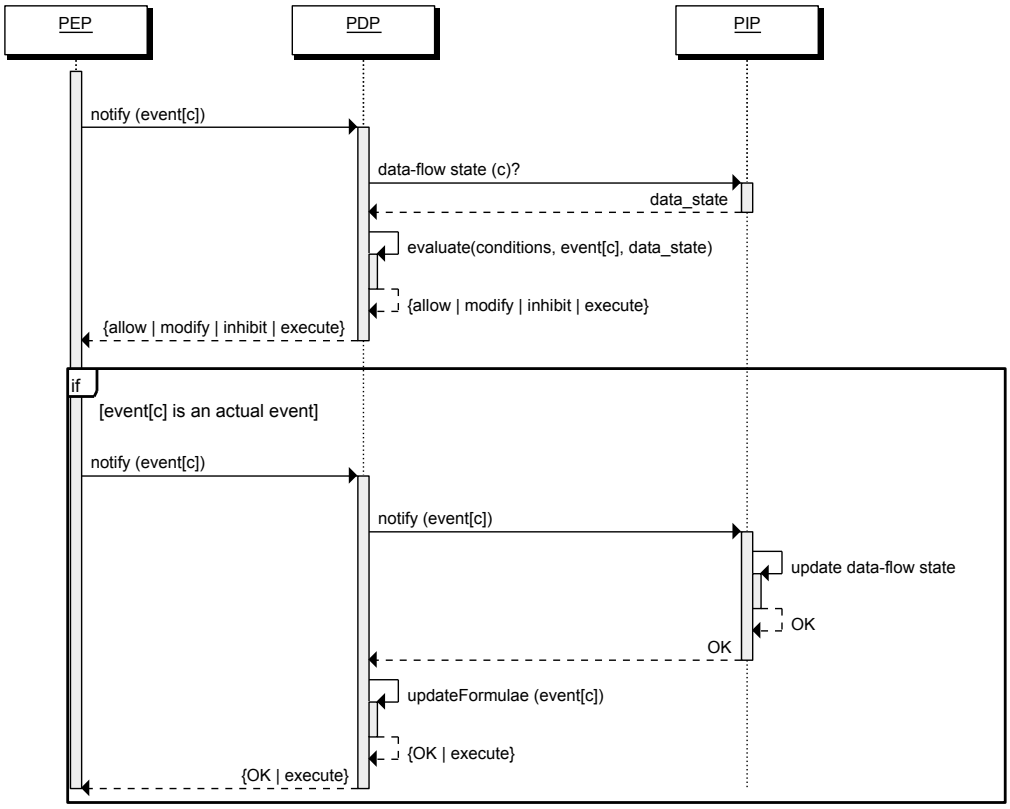


Figure 5.2.: Single-layer behavior [137].

In these cases the PDP queries the PIP for information about data dissemination, e.g. to know the relationship between a certain data item and its representations.

Finally, if the desired event is transformed into an actual event and executed by the PEP, the PEP notifies the PDP about the execution, which then updates corresponding formulae. The PDP in turn, propagates the notification about the executed actual event to the PIP, in order to allow it to update the data flow model accordingly. The evolution of the data flow model is internally performed by updating the data state $\sigma \in \Sigma$ according to the semantics of the executed event, defined within the transition relation ϱ. Figure 5.2 depicts the corresponding sequence of interactions.

The runtime verification of mechanisms within the PDP are realized on the basis of tableaux. In particular, the implementations described in these pages implement the algorithm proposed by Rosu, Havelund, and Geilen [74], which only maintains the current state of a formula, thus allowing for runtime verification without the need of storing the complete event history. The data flow tracking functionality of the PIP is implemented as a state machine, whose state transitions are triggered by the evaluation of the semantics of incoming events. More specifically, the formal semantics is translated into sequences of updates of the storage, alias and naming function of the model. These updates are then performed on the basis of the name and the parameter values of the actual events the PIP receives from the PDP.

Multi-layer

The architecture of a multi-layered system corresponds to the architecture for the single layer, with the only difference that there exists a monitor (i.e. a PEP) for each layer of abstraction (see Figure 5.1b). All the different PEPs signal events to the same PDP. Because of the likely different notion of timestep at different layers, some minor assumptions needs to be taken about the monitors, the speed at which they notify events to the PDP and the possible interactions between the layers. All of these assumptions are discussed in detail in Chapter 4, together with a description of the respective PIP.

These differences, however, do not require any change in the architecture: the sequence diagram describing the components interactions in this case would be identical to the one depicted in Figure 5.2 for the single case, where the PEP agent represents any of the multiple PEPs in the system.

Multi-system

The model defined in Chapter 4 for the tracking of data across two layers of abstraction does not require the two layers to be part of the same physical machine. This implies that, at least in principle, the work described in this dissertation could also be applied out-of-the-box to distribute systems. However, while at the model level the approach is conceptually the same, technically it is fundamentally different.

Distributed systems introduce a number of challenges to usage control and data flow tracking, both conceptual (e.g. taking decisions if the network is down and the PDP is unreachable) and technical (e.g. a centralized PDP requires every PEP to notify every event remotely, generating a huge amount of traffic). For this reason, this work focuses on single physical systems only. Additional challenges introduced by the distributed aspects are addressed by related work in the literature [61, 91, 92].

5.2. Implementation and Evaluation

In contrast to the Policy Enforcement Point, which is bound by its very nature to a specific layer of abstraction, the Policy Decision Point and the Policy Information Point components in this architecture could be implemented in a layer-agnostic manner and shared by PEPs at any layer. During the development of the research presented in this thesis, instantiations of the data usage control framework have been implemented and evaluated for different layers of abstraction, [55, 91, 100, 97, 27, 26, 163, 105, 165] including the two examples described in Section 3.3 and Section 4.4 [164, 149], often by students in the context of their Bachelor and Master theses under the author's supervision.

In terms of performance, PEPs are usually implemented in form of reference monitors [51]. Depending on the layer of abstraction considered, the overhead such monitors introduce ranges from 2-3 orders of magnitude [136] to almost negligible under normal usage (i.e. no stress testing) [55]. The performance of the PDP and PIP implementations depends on multiple factors, like number of policies, frequency of events, number of parameters of the events, communication interface (network vs direct function call), amount of data elements already present in the system, etc. With few exceptions, like the model for quantitative data flow tracking described in Chapter 8, the overhead induced by PDP and PIP

computation is in general negligible compared to the overhead induced by the PEP. Some examples of implementations are discussed in the evaluation sections of the models in the second part of this work (Section 6.5,Section 7.3 and Section 8.5). More detailed evaluations for specific layers of abstractions can be found in the respective literature (see Chapter 9).

While a discussion about formal soundness of the generic model is provided in Section 3.4 and Section 4.2, correctness of all the different implementations have always been shown in the literature by case studies only. The general idea is to provide some exemplary traces of events (like "⟨ copy A to B, delete A ⟩"), the result of the execution of which is intuitive (e.g. "A is deleted and B contains what A contained before the execution"), and then verify that the model is in the correspondent states after processing the same traces.

A common source of problems for any implementation of the generic model for data flow tracking is the so-called *label creep* issue, described below.

5.2.1. Label Creep

Data flow tracking is based on taint analysis. The general rule of taint analysis is

> *"the result of an operation should be marked with all taint marks of its operands"*.

In the terms used in the rest of this work, this conveys the point that if any of the operands contains certain sensitive data, after the execution of the operation such data is possibly stored in the result of the operation too.

Taint analysis is composed by three phases:

- Initial *classification* of containers;
- *Propagation* of data according to the general rule described above and
- *Declassification* of containers, e.g. if all content of a container is deleted.[1]

In the literature, like in this framework, data flow tracking has been used to support security and therefore the analysis results tend to be conservative, i.e. reporting false positives. Such overapproximations cumulate over time and lead to the so-called *label creep* situation, in which all taint marks are associated with many system containers. In this context, further data flow tracking becomes pointless because, according to the analysis, data is already stored everywhere; moreover, if the usage of tracked data is constrained by policies, even the system's stability might get compromised, because every marked container, actually containing the data or not, would be subject to the constraints.

The second part of this work presents different models for data flow tracking that aim at mitigating label creep by augmenting the framework with additional information about the system.

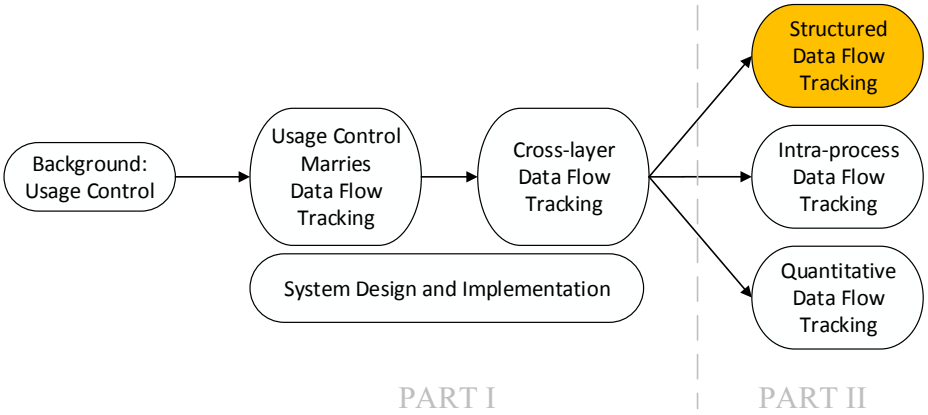
[1] A data item represents a policy that imposes restrictions on the usage of data, therefore *classification* corresponds to storing data into a container and *declassification* corresponds to removing it.

Part II
Taming Label Creep: Enhanced Data Flow Tracking

*The second part of this work discusses different enhanced models for data flow tracking. In the following, the term "**basic model**" refers to the (data flow tracking component of the) model described in the first part. The goal of the solutions described in this part is an improvement in terms of analysis precision w.r.t to the basic model. For each approach presented, assumptions, limitations, strengths and drawbacks are discussed. While some of these solutions could in principle be combined and applied at the same time, each approach is evaluated* individually *at the end of the respective chapter for a better understanding.*

6. Structured Data Flow Tracking

This chapter describes an extension of the model in which the tracking precision is improved using information about the structure of data. A preliminary version of this work has been published in [108], a publication co-authored by the author of this dissertation. The implementation and evaluation of the model are part of this thesis's original contribution.



6.1. Introduction

The model for cross-layer data flow tracking presented in Chapter 4 shows how to combine data flow tracking at different layers of abstraction to obtain a more precise estimation of the data dissemination within the system. The intuition is that, given an instantiation of the model for a layer of abstraction, it is possible to leverage additional information about the system (in that case, a monitor for another layer) to improve the precision of the tracking.

This chapter exploits the same fundamental intuition to refine the precision of the tracking, in this case by leveraging information about the *structure* of data. In those scenarios where such additional information is available this solution can help mitigating the problems introduced by an overly-conservative data flow analysis (*label creep*, see Section 5.2.1). The fundamental research questions addressed in this chapter are

How can the precision of the data flow tracking model be augmented with additional information about the structure of data?

How can the improvement in terms of precision be quantified?

What is the overhead induced by this approach?

Which aspects of the system influence the results?

6.1.1. Bottle-neck Pattern

Practical experience with the instantiation of the basic model at different layers in real systems suggests that a common source of overapproximations is the *bottle-neck pattern*, depicted in Figure 6.1: If the content of multiple containers with different data is merged into a single container, then the resulting container (*intermediate container*, from now on) correctly stores all of the data in its sources. By applying the basic rule of taint propagation ("the result of an operation should be marked with all taint marks of its operands", see Section 5.2.1), further operations on this intermediate container unconditionally propagate all these data items, often leading to label creep.

Scenarios in which the bottleneck pattern can be observed include but are not limited to: saving/loading application-specific content (multiple containers) to/from one file, compression of several files into one archive and subsequent decompression, copy-pasting data via the clipboard, and transferring content via pipes or sockets. In all of them, each destination container (like a file extracted from an archive) should only contain the data initially stored in one specific source (the same file before compression), rather than all data items in the intermediate container (archive file).

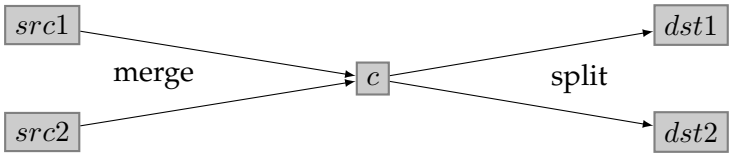


Figure 6.1.: Bottle-neck pattern [108].

The solution presented in this chapter builds on one central observation: in all of these scenarios there exists a pair of dual *merge* and *split* operations. While a *merge* operation (such as *zip*) aggregates content from different sources into one intermediate container, like c in Figure 6.1, the corresponding *split* operation (such as *unzip*) reads the same content and separates it into multiple containers, each matching exactly one of the source containers.

6.1.2. Proposed Solution

The solution proposed here is a generic model for data flow tracking that, in correspondence of a merge operation, stores into the intermediate container a special *structured data item*. Such data represents all data items of the source containers, but stores additional information concerning which source contained which one, like a snapshot of the storage function (see Section 3.1) for those containers. This information is then exploited by split operations to propagate only selected data to each destination, thus effectively *declassifying* all the other destination containers. This mitigates the label creep issue by decreasing the amount of false positives.

Consider the example depicted in Figure 6.2: $file_1$ and $file_2$, respectively containing data d_1 and data d_2, are archived into a file named $archive.zip$. When the archive is extracted, a flow of data from $archive.zip$ to two files ($file_1'$ and $file_2'$) takes place. Each of the two files is a copy of the respective source. The idea behind the structured taint mark is to associate to $archive.zip$, during compression, a new taint mark d_{str} that maps $file_1$ to d_1 and $file_2$ to d_2; during extraction, such mapping is used to associate to $file_1'$ ($file_2'$) only data d_1 (d_2),

instead of both d_1 and d_2, or even d_{str}, as a straightforward application of the basic taint propagation rule would suggest.

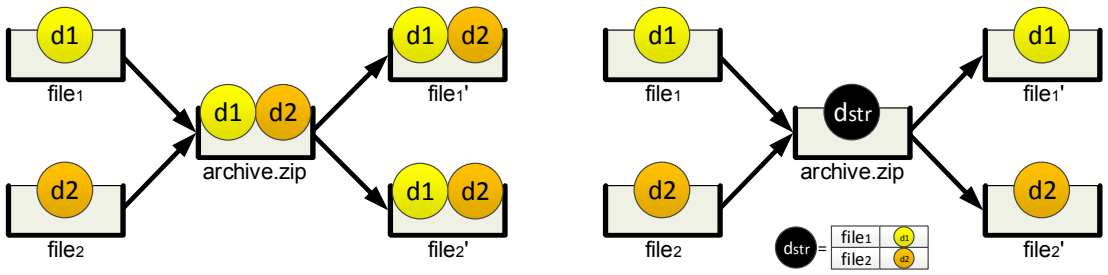


Figure 6.2.: Example of data flow tracking without (left) and with (right) structured data flow tracking (Packing and extracting an archive file).

Note that in many scenarios, especially when data flows across different layers of abstraction and thus its representation changes "format", a trivial solution to the problem without the use of structured taint is to encode the data items, i.e. metadata, as additional content within the container, e.g. into the file where an application is saving, and to reuse this information at loading time. This way, the propagation of the taint mark for certain content will be carried out automatically by the events that transfer the content itself.

However, data flow tracking analysis should interfere as little as possible with the original behavior of the system, and because of this, the proposed solution achieves the same result in a *transparent* way, never changing the actual content being transferred by a system event. The motivation behind this choice is twofold: on one hand, changing the content may compromise the integrity of a container or make it unusable (e.g. by invalidating its signature); on the other hand, this solution is generic enough to be applicable at any layer and does not depend on nor is constrained by the technical representation of a container. With respect to existing solutions that pursue this approach (see Chapter 9), this approach does not require additional number or types of events, finer granularity of containers, nor qualitative/semantic analysis of the transferred content.

Also note, that the precision of the analysis for the intermediate container is equivalent to that of basic taint propagation (see Section 6.3.2); the reduction in terms of false positives is obtained only in the destination containers of the split operation.

Before performing *declassification*, however, it is important to make sure that the structured data item associated with an intermediate container is *valid*: the structured data must not have been propagated due to overapproximations, and the integrity of the intermediate container's content must be assured; if this is not the case, the analysis falls back to basic taint propagation. To this end, additional integrity checks are performed (see Section 6.3.3).

Problem. This chapter tackles the problem of label creep in taint-based data flow tracking analyses.

Solution. The proposed solution is a generic model for taint-based data flow tracking of structured data, that can be instantiated for different contexts at different system layers.

Contribution. The contribution of this work is the first generic solution for event-based structured data flow tracking. This model *transparently* builds, propagates, and uses special data items that reflect the inherent structure of data without semantic analysis of the tracked content. With minimal assumptions on the system and without modifying the number or the granularity of events or containers, this model can increase the precision of existing data flow tracking analyses and mitigate the label creep problem.

Assumptions. In addition to the usual assumptions (see Section 1.3), this model requires the existence of dual merge and split events as explained above. These events

- must be detectable and identifiable at runtime,
- must have clearly defined semantics, in particular in terms of data propagation,
- must be trusted, i.e. there must exist confidence that they behave according to the expected semantics.

6.1.3. Example Scenario

> *Alice works as an analyst for a smartphone manufacturer. Alice's duties include gathering information about new models under development and combining it with data from field experiments and from various public sources into reports for suppliers and for other departments. Reports are prepared in Excel, and* ***every exchange of data across departments happens in form of archived files (e.g. zip archives)****. In addition to the usual security precautions, like forbidding the installation of third-party software, each enterprise machine is equipped with an OS-layer monitor that tracks the execution of every system call and blocks/reports those attempting to send sensitive data to unauthorized remote addresses. The OS model rely on a basic taint propagation approach for the tracking that usually performs pretty well, despite introducing minor* ***over-approximation****; sometimes, however, it* ***creates some trouble in the collaboration*** *between departments. Whenever Alice sends* ***an archive containing a set of reports*** *to be added to the repository, if* ***one of the reports is protected*** *by the policy "*don't send to ThatCompany inc.*",* ***none of the reports*** *within the same archive* ***can be sent*** *to that particular destination. Similarly, if Alice* ***protects one sheet within a workbook*** *with the policy "*this sheet should not be printed*", when Bob receives the file* ***no sheet can be printed****.*

Consider Alice opening two files, $rep1.xls$ and $rep2.xls$, respectively containing workbooks $Book1$ and $Book2$. Assume $rep1.xls$ is protected by the "*do not send to X*"-policy. The cross-layer model presented in Chapter 4 can be used to propagate this restriction only to $Book1$ and not to $Book2$. Similarly, when Alice saves a workbook as $rep3.xls$, the cross-layer model can be used to distinguish whether Alice is saving $Book1$ or $Book2$, and consequently propagate the policy to $rep3.xls$ only when necessary. In a nutshell, the cross-layer model can be used in this context to distinguish one file from the other within Excel.

Assume now that Alice edits $Book2$ and specifies a policy for cell $B3$ on sheet $Sheet3$ that stipulates that if the document is printed, the content of such cell should be replaced

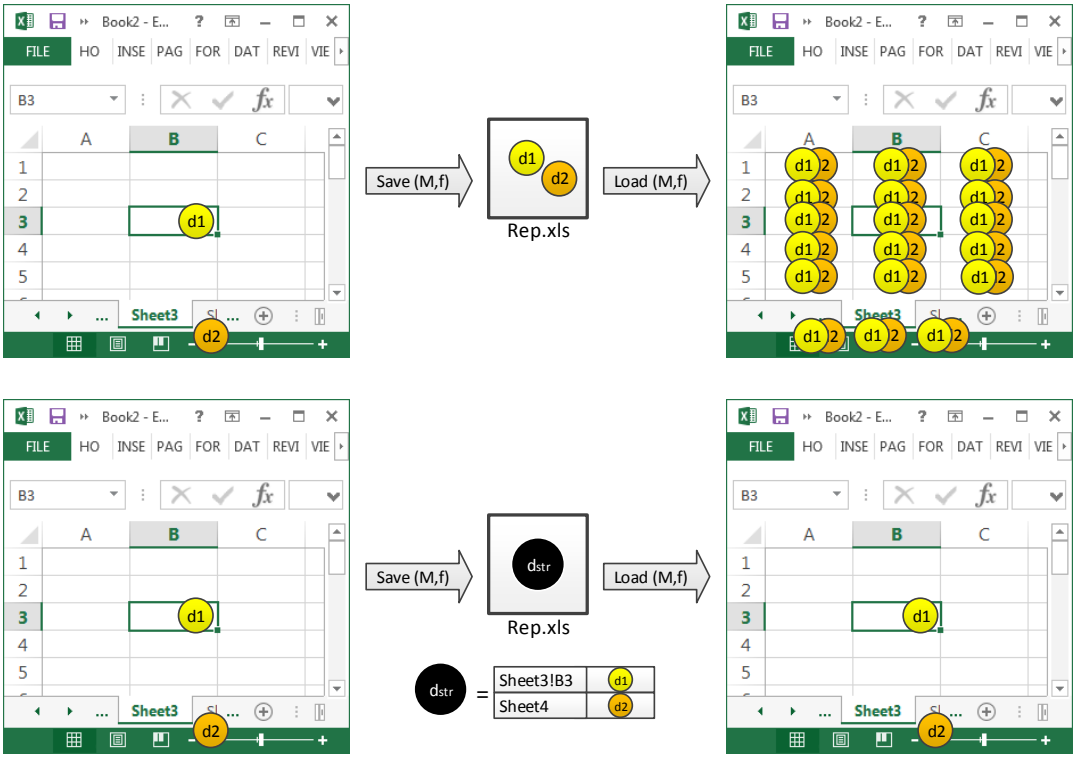


Figure 6.3.: Example of data flow tracking without (top) and with (bottom) structured data flow tracking (Saving and re-loading a file with MS Excel).

by a black rectangle (Modifier ILP, see Section 2.3), and a policy for sheet *Sheet4* stating that the content of *Sheet4* should not be disclosed over the network. Let associate the first policy to data item *d1* and the second policy to data item *d2*. When Alice saves *Book2* into *rep.xls*, the monitor for Excel should propagate *d1* and *d2* to the monitor for the OS, which in turn associates them to *rep.xls*. The problem is that the OS monitor does not "understand" abstractions like "cell" or "sheet", which are proper of the Excel domain, and therefore associates *d1* and *d2* to the complete file. Although this is sound from a security perspective (it is a conservative overestimation of the content to be protected), the mapping between the data items and the specific parts of the document is lost. When the file, or a copy of it, is opened again in Excel, in order to stay conservative the monitor for Excel needs to propagate the taint marks *d1* and *d2* to *any container associated to the loaded file*, i.e. to any container in *Book2* (Figure 6.3, top).

In summary, the improvement provided by the cross-layer model in this example is bounded by the granularity of the tracking at the OS layer.

The same identical problem can be observed when multiple reports, marked with different policies, are stored in a single zip file. Also in this case the OS monitor is not capable of distinguishing between different parts of the archive and therefore associates the whole

file to any data stored in the source files. When the archive is extracted, every destination file is marked with all the taint marks of the archive file, similarly to what happens in the Excel scenario (see Figure 6.2, left).

Leveraging structured taint marks, in contrast, the intermediate representation can preserve storage information about the sources and use it to declassify the destination in case of split operations (Figure 6.2, right and Figure 6.3, bottom).

6.2. Formal Model

The model for structured data flow tracking (from now on called *structured model*, for brevity's sake) is an extension of the basic model presented in Chapter 3. In the structured model, a system is described as a tuple

$$(\mathcal{D}, \mathcal{S}, \mathcal{C}, \mathcal{F}, V, partId, checksum, \Sigma, \sigma_i, \mathcal{R})$$

where $\mathcal{D}$, $\mathcal{S}$, $\mathcal{C}$ and $\mathcal{F}$ represent, respectively, the sets of data items, system events, containers and container labels of the basic model. In this model, $\mathcal{F}$ also contains a set $\mathcal{F}_P \subseteq \mathcal{F}$ of identifiers for parts of data structures (explained in more detail in Section 6.3), also called partIDs.

$partId : \mathcal{C} \mapsto \mathcal{F}_P$ assigns partIDs to containers. $partId$ is used in split operations to decide which part of a structured taint mark corresponds to a destination container. In the model $partId$ is an oracle, while Section 6.4 describes possible instantiations of it.

$checksum : (\mathcal{C} \times \mathcal{D}) \to V$ is another oracle that computes a checksum of the content of a specific container (e.g. the hash of a file), with V the set of all possible checksum values.

In addition to the storage, alias and naming functions (see Section 3.1), a state of the system $\sigma = (s, l, f, struct, checkList)$ is also defined by

- a *structure function* of type $struct : \mathcal{D} \to \mathbb{P}(\mathcal{F}_P \times \mathbb{P}(\mathcal{D}))$, mapping data items to a structure. A data item associated to a non-empty structure is called structured data or structured taint mark;
- a *checklist* $checkList \subseteq \mathbb{P}(\mathcal{D} \times V)$, used to check whether the structure associated to a certain taint mark is valid.

Like in the basic model, Σ is the set of all system states, including the initial state σ_i and $\mathcal{R} \subseteq \Sigma \times \mathcal{S} \to \Sigma$ is the transition relation that encodes how a state $\sigma \in \Sigma$ must be updated in case an event occurs.

6.3. Structured Data Flow Tracking

Sometimes data presents an inherent structure: a mail has a recipient, a subject, and a body; a story is divided into chapters and sections; a song into chorus and verses, etc. Although sometimes this structure is reflected by the container in which the data is stored, conceptually it remains a property of the data rather than the container. Data structure is preserved even when its concrete representation is "obfuscated", e.g. by means of compression. For this reason, the structured model binds the structure to the data item (i.e. the taint mark) rather than to the container.

More precisely, if some data carries an inherent structure, the relative taint mark is associated with a set of partIDs $\{partID_1, \ldots, partID_n\} \subseteq \mathcal{F}_P$, each of which is in turn associated with a set of taint marks. The rationale is that these partIDs identify the different parts of the structured data (a certain worksheet or a certain cell in the example in Figure 6.3) whilst the associated taint marks represent the data items associated with the corresponding part (e.g., *d2* for 'Sheet4'). Formally, the relation between each taint mark and its structure is given by the function

$$struct : \mathcal{D} \rightarrow \mathbb{P}(\mathcal{F}_P \times \mathbb{P}(\mathcal{D})).$$

For any unstructured data d, $struct(d) = \emptyset$.

Associating the structure with the taint mark rather than a container presents the advantage that the taint mark is propagated independently of the type of containers in which the content is actually stored and independently of whether operations on such containers are aware of the structure or not. This allows for easily reuse data flow event semantics from Section 3.3 and from other work from the literature [138, 136, 164, 100, 91, 73], i.e. to seamlessly integrate with existing instantiations at different system layers. Only the semantics of those events that correspond to merge and split operations need to be updated.

Note that the structured model supports the nesting of structured taint marks. Also note, that if a container c is marked with a structured taint mark d, with $\sigma.struct(d) = \{(p_1, \{d_1\}), (p_2, \{d_2\})\}$, then both restrictions imposed by d_1 and d_2 apply to c.

Let $flat : \Sigma \times \mathcal{D} \rightarrow 2^{\mathcal{D}}$ be an operator that returns the set of atomic (i.e. non-structured) data items stored within a certain structure (the one associated to the given data item). *flat* corresponds to the transitive closure application of the *struct* function, and is defined as

$$\begin{aligned}\forall \sigma \in \Sigma, d \in \mathcal{D} : flat(\sigma, d) = \{d' \in \mathcal{D} | (d' = d \wedge \sigma.struct(d') = \emptyset) \vee \\ (\exists f \in \mathcal{F}_P, D \in \mathbb{P}(\mathcal{D}), d'' \in \mathcal{D} : (f, D) \in \sigma.struct(d) \wedge d'' \in D \wedge d' \in flat(\sigma, d'')\}\end{aligned}$$

For instance, for the examples described in Figure 6.2 and Figure 6.3, $flat(\sigma, d_{str}) = \{d1, d2\}$, with σ being any state right after the creation of the intermediate container.

The *flat* operator can be used to adjust the notion of event refinement given in Section 3.2 to cope with structured data, by replacing the check $d \in \sigma.s(c)$ with $d \in \cup_{d' \in \sigma.s(c)} flat(\sigma, d')$. Note that the first part of the definition of the *flat* operator implies that for any non-structured data item d in state σ, $flat(\sigma, d) = d$, and, consequently, that for any container c that does not contain any structured data item, $\cup_{d' \in \sigma.s(c)} flat(\sigma, d') = \sigma.s(c)$.

$$\begin{aligned}&\forall e_1, e_2 \in \mathcal{S}, \forall \sigma \in \Sigma : (e_1, \sigma)\ refinesEv_i\ e_2 \iff \\ &\quad (getclass(e_1) = getclass(e_2) \wedge e_1\ refinesEv\ e_2) \vee \\ &\quad (getclass(e_1) = containerUsage \wedge getclass(e_2) = dataUsage \wedge \\ &\qquad e_1.n = e_2.n \wedge \exists d \in \mathcal{D}, \exists c \in \mathcal{C} :\ d \in \cup_{d' \in \sigma.s(c)} flat(\sigma, d') \wedge \\ &\qquad obj \mapsto c \in e_1.p \wedge obj \mapsto d \in e_2.p \wedge e_2.p \setminus \{obj \mapsto d\} \subseteq e_1.p \setminus \{obj \mapsto c\}).\end{aligned}$$

6.3.1. Merge Operations

In some scenarios it is possible to contain label-creep by leveraging additional information about merge operations. In the terminology of this work, merge operations are special system events that

- aggregate data from multiple sources into a single destination container,

- have corresponding dual *split operations*, and
- allow us to infer information about the structure of data.

The latter usually comes from external knowledge about the system, e.g. the fact that process 'zip' is an archiver. The inferred structure is associated with a new structured taint mark for the destination container (i.e. the intermediate container in a bottleneck pattern).

Formally, a merge operation $mo(SRC, dst)$ merges the content of the set of source containers $SRC \subseteq \mathcal{C}$ into the single destination container $dst \in (\mathcal{C} \backslash SRC)$ in a structured way. This means that all taint marks associated with all source containers are grouped into multiple (possibly overlapping) sets, each of which is identified by a partID. The partIDs are derived by some properties of the set SRC, e.g. the name of the containers, and they are captured by the layer-specific function $partId : \mathcal{C} \mapsto \mathcal{F}_P$. At the model level, $partId$ is the oracle that determines which parts of a structured taint mark correspond to which destination container of a split operation. While the implementation of $partId$ is instantiation-specific (see Section 6.4 for some examples), the semantics of any *merge* event mo can be described in a generic way as

$$
\begin{aligned}
&\forall \sigma, \sigma' \in \Sigma, \forall SRC \subseteq \mathcal{C}, \forall dst \in \mathcal{C} \backslash SRC : (\sigma, mo(SRC, dst), \sigma') \in \mathcal{R} \implies \\
&\quad \sigma'.s(dst) = \sigma.s(dst) \cup \{d_{str}\} \wedge \\
&\quad \sigma'.l = \sigma.l \wedge \\
&\quad \sigma'.f = \sigma.f \wedge \\
&\quad \sigma'.struct(d_{str}) = \{(partId(c), \sigma.s(c)) | c \in SRC\} \wedge \\
&\quad \sigma'.checkList = (\sigma.checkList \setminus \{(d_{str}, v) | v \neq checksum(dst, d_{str})\}) \\
&\qquad\qquad \cup \{(d_{str}, checksum(dst, d_{str}))\}
\end{aligned}
$$

where d_{str} represents a previously unused data item, i.e. such that $\forall c \in \mathcal{C}, \forall d \in \sigma.s(c) : (d_{str} \neq d) \wedge (d_{str} \notin flat(\sigma, d))$, and $\sigma.s$, $\sigma.l$, $\sigma.f$, $\sigma.struct$ and $\sigma.checkList$ indicate, respectively, the storage, alias, naming and structure functions, and the checklist of state σ. Performing a merge operation requires updating the list of valid checksums, which is why the old checksum values of (dst, d_{str}) are replaced with the new checksum value.

While merge operations are implicitly assumed to be atomic, in real systems a merge operation might be composed of multiple subsequent events. In this case, the structure must be built incrementally; this more complex case is not discussed in this work (see Section 6.6 for more details).

6.3.2. Split Operations

A split operation $so(src, DST)$ is the dual of a corresponding merge operation. It propagates the content of one source container $src \in \mathcal{C}$ to a set of destination containers $DST \subseteq \mathcal{C}$. In the bottleneck pattern, the source container src corresponds to the intermediate container. In contrast to normal taint propagation events, split operations leverage the fact that the source container is marked with a structured taint mark. As this structure was built based on information about the corresponding merge operation, split operations use this additional information to declassify the destination containers DST. In other words, split operations do not necessarily propagate all taint marks associated with the source container src to all destination containers DST, but only selected taint marks to selected containers.

For this reason, split operations do not follow the conservative approach of overapproximating data flows. Instead, they perform selected declassification of the destination containers, thus mitigating the label creep problem. Which taint marks are in fact propagated to which destination containers is determined by the application of the $partId$ function to each destination container. If the result is such that a corresponding partID exists in one of the source container's structured taint marks, only the taint marks related to this partID are propagated; if no such match is found, all taint marks are blindly propagated, which is equivalent to basic taint propagation. Formally:

$$
\begin{aligned}
&\forall \sigma, \sigma' \in \Sigma, \forall src \in \mathcal{C}, \forall DST \subseteq \mathcal{C} \backslash \{src\} : (\sigma, so(src, DST), \sigma') \in \mathcal{R} \implies \\
&\quad \forall dst \in DST : \sigma'.s(dst) = \sigma.s(dst) \\
&\qquad \cup \{d' \in \mathcal{D} \mid \exists d \in \sigma.s(src), \exists D' \subseteq \mathcal{D} : \ (partId(dst), D') \in \sigma.struct(d) \wedge \\
&\qquad\quad (d, checksum(src, d)) \in \sigma.checkList \wedge d' \in D'\} \\
&\qquad \cup \{d \in \sigma.s(src) \mid \ \nexists D' \subseteq \mathcal{D} : (partId(dst), D') \in \sigma.struct(d) \wedge \\
&\qquad\quad (d, checksum(src, d)) \in \sigma.checkList\} \wedge \\
&\qquad \sigma'.l = \sigma.l \wedge \\
&\qquad \sigma'.f = \sigma.f \wedge \\
&\qquad \sigma'.struct = \sigma.struct \wedge \\
&\qquad \sigma'.checkList = \sigma.checkList
\end{aligned}
$$

Note that the last part of the definition of the storage component guarantees that if $(d, checksum(src, d))$ is not in $\sigma.checkList$ the integrity of the source container has been compromised and thus the model falls back to basic taint propagation.

Also note that the enhancement in terms of precision is only achieved at the time of the split operation. As already mentioned in Section 6.1.2, whenever a restriction is associated with a taint mark d, it applies to every container c that is marked with d, either directly ($d \in \sigma.s(c)$) or indirectly ($d \in \mathit{flat}(\sigma, d') \wedge d' \in \sigma.s(c)$).

6.3.3. Checksum

Given a certain container $c \in \mathcal{C}$ and a structured data item d, $checksum : (\mathcal{C} \times \mathcal{D}) \to V$ computes a value that is used to decide whether the structure of d is reliable for the content of c or not. This value is called *checksum* and V is the set of all possible checksum values. In some cases, the structure is valid only if the content matches exactly the content for which the structure has been created (e.g. in the archiver case, the hash of a file), which makes the result of $checksum(c, d)$ independent of the value of d.

In other cases the content may be changed, usually augmented, and still be valid. In these cases, the checksum values depend only on those parts of the content that are listed in the structure of d. For instance, if the content is an XML document, then the checksum will look at the integrity of only those nodes mentioned in the structure. For parts that are not mentioned in the structure, or, in general, if the checksum-check fails, the conservative "propagate-everything" approach applies. A generic hash function, like MD5 or SHA1, is in general a good choice for a $checksum$ function, although, depending on the considered scenario, sometimes it may be too restrictive.

The list of valid checksums is stored in a relationship called $checkList$. Given a container, a data item and a checksum for its structure, $checkList$ returns whether the checksum is

valid or not. Note that although the parameter of $checksum$ is in $\mathcal{C}$, the function is computed on the *actual content* of the container, and thus its definition is layer specific.

6.4. Instantiations

The model described so far is applicable to any scenario similar to the Excel example (see Section 6.1.3), where application-specific data flow tracking is combined with tracking at the operating system layer. However, there are more situations in which the structured model can improve the precision of basic taint propagation.

Consider the action of copy-and-pasting multiple data within an application. Although the system clipboard preserves the structure of the content, if the clipboard is modeled as a single container [105, 165], it behaves as an intermediate container in the bottleneck pattern and propagates all taint marks of the sources to all destination containers. In this case the event "copy" ("paste") corresponds to a merge (split) operation. The instantiation of $partId$ is application-specific, e.g. for a spreadsheet application it would map the cells to their 'coordinates', while for a word processor it would work on internal identifiers of sections, paragraphs, or words.

Similarly, consider a data flow tracking analysis for the operating system [73], where containers are files, pipes and memory locations, and events are system calls; unless there exists a dedicated monitor for the application, whenever a process reads from a file, its memory gets associated with all the data stored in that file. Such data is then be propagated to every file the process writes. However, in the special case of an *archive* process such as 'zip', the extraction (split) of an archive propagates to each extracted file only the data items that were associated with it at the moment the archive was created (merge), rather than all taint marks associated with the archive file, as depicted in Figure 6.2. Function $partId$ maps source and destination containers using their relative filenames. In Section 6.5 an instantiation of the model for this particular case is implemented and evaluated.

The structured model also applies in distributed scenarios, e.g. when many files are transferred over a TCP connection. While existing solutions [91] address the problem of how to propagate the taint marks from one system to another, the communication channel behaves like the intermediate container in the bottle-neck pattern. In this scenario, the merge operation is the sequence of read (from files) and write (to the socket) events observed on the "sender" side, whereas the split operation corresponds to the dual sequence on the "receiver" side. In a simple file transfer scenario, function $partId$ would map source and destination containers using filenames.

In all these examples, a simple hash of the content or a set of hash values for the different parts of the content could be used for the checksum function, to guarantee that the to-be-split content exactly matches the one at the time of merge.

6.5. Evaluation

This section describes the experiments conducted to evaluate the improvement in terms of precision offered by the structured model and the relative cost in terms of performance loss. These experiments address the following research questions:

RQ1 How much more precise is the structured approach with respect to the estimation provided by the basic model?

RQ2 How much slower is the structured model with respect to the basic model?

A first preliminary test based on the implementation of the Excel example (see Section 6.1.3) is presented in see Section 6.5.1. Afterward, the settings of the systematic set of experiments used for the evaluation are illustrated in Section 6.5.2, elaborating on the problem of monitoring non atomic events in Section 6.5.4. The experiments addressing the two research questions are then described in Section 6.5.5 (**RQ1**), and in Section 6.5.6 (**RQ2**). Finally, the results of the experiments are discussed in Section 6.6.

6.5.1. Preliminary Test

In order to test the improvement in terms of precision of the structure data flow tracking, the scenario described in Section 6.1.3 has been implemented. The implementations for the usage control framework for Microsoft Excel [149] and for Microsoft Windows [164], taken from the literature, are described in Section 3.3. Both implementations have been connected to the same PDP and to the same PIP extended with structured data flow tracking concepts, with `SAVE()` and `LOAD()` events configured as Merge- and Split-operations, respectively.

As expected, in correspondence of a `SAVE()` event, a new structured data item was created in the model, containing all the data stored in the Excel workbook, with the respective references. The structured data-item was then stored in the container representing the destination file according to the cross-layer model. From then on, operating system events that copied, moved or deleted the file or one of its copies were detected by the operating system monitor and modeled accordingly. Any further `LOAD()` event was then modeled as a cross-layer split-event, correctly restoring the mapping between each cell of the opened document and the respective data.

However, because the Excel monitor only intercepts user events (like clicking on the save button, or pressing CTRL+C), it turned out to be a non-trivial effort to *automate* the simulation of such events, in particular for performance evaluation. For this reason, and to prove the general applicability of the solution to different contexts, the structured model has been systematically tested in different settings.

6.5.2. Experiment Settings

The experiments described in this section are executed on a system with a 2.3GHz Intel Core 2 Duo E6400 and 2GB of RAM running a vanilla version of Linux Mint 17 [7]. The implementation of the Policy Enforcement Point (see Section 5.1.1) is taken from the literature [91], and is based on a modified version of the strace tool [9]. The tests have been performed executing the rest of the usage control infrastructure on the same machine and connecting the components using the Thrift protocol [56] via a local network connection.

6.5.3. Experiment Description

In the scenario discussed in Section 6.1.3, Alice exchanges reports with other departments in form of archive files, e.g. zip files. A typical use case is that Alice takes some of the reports that she has recently been working on, compresses them into a zip file, and send them to the concerned department, where another clerk unzips the archive and makes use of its content to create other reports.

The instantiation of the structured model at the operating system layer models every creation of an archive using the zip command as a merge event and every invocation of the unzip command as the respective split event.

The experiment starts from a state in which an initial set of ten files, representing ten sensitive reports, has been created. Each file is composed by 100 identical lines (identical to each other and different from those in other files), each containing the identifier of the sensitive data contained in that specific document, e.g. "Data01".

A script then generates a set of random traces of events from the following classes:

- $zip\ file_{dst}\ file_{src_1}...file_{src_n}$, to create a new archive called $file_{dst}$ containing a copy of all the files $file_{src_1}...file_{src_n}$;
- $unzip\ file_{src}\ file_{dst_1}...file_{dst_n}$, to extract $file_{dst_1}...file_{dst_n}$ from the archive $file_{src}$;
- $copy\ file_{src}\ file_{dst}$, to create a new copy of $file_{src}$ called $file_{dst}$;
- $delete\ file$, to delete a file called $file$;
- $integrate\ file_{dst}$ N $file_{src_1}...file_{src_n}$, to integrate an existing report with some information from other reports. In particular, for the simulation, such command is implemented as a small executable that reads N random lines from each source file $file_{src_1}...file_{src_n}$ and appends them to $file_{dst}$.

Note that only valid traces are generated: for instance, if a trace contains an $unzip$ event at time t, then executing the trace from the beginning until time $t-1$ starting from the initial state with the ten files described before, it must be the case that at time t there exists at least an archive that has been created and not deleted; similarly, if the state at time $t-1$ results in only one file in the system, then the event at time t can not belong to class $delete$.

Each trace is executed in the monitored system and its execution time recorded for performance evaluation (see Section 6.5.6). The list of unique lines within each file is also stored and compared with the estimation provided by the structured model and by the basic model like the one presented in Section 3.3.2 (see Section 6.5.5).

Due to the lack of real world usage profiles, the creation of the traces is randomized assigning a different probability to each of the five event types. This probability reflects the frequency with which events from a certain class are likely to appear in the final trace. Additionally, other aspects of the traces have been parametrized, like the maximum number of sources in a merge events, the size of the initial files, the number of lines being merged by $integrate$ events, etc. For each combination of parameters, 100 different traces have been generated and only average values and standard deviation are reported, in order to factor out random noise.

Depending on the goal of the measurements (precision vs performance), different sensitivity tests have been performed to identify which variables have a direct impact on the results. The outcome of these tests are discussed in the respective sections.

6.5.4. Handling of Non-Atomic Events

Because a model for non-atomic split and merge events is still work in progress (see Section 10.1), in the experiments, zip and unzip commands have been modeled in a special way, similar to atomic events.

In correspondence of merge events, the structured data was built upon observing the respective `execve` event, one of the first system calls triggered by any command. The `execve` system call contains the complete list of command line parameters used to invoke the process, and thus it contains all the information needed to build the structured data. This imposes the further assumption that data in the source files does not change in-between the moment in which the `execve` system call is observed and the last source is read; although it may sound restrictive, this behavior is actually enforced by the application itself, which fails if any input file is modified during the archiving process. Once the structured data item d_{str} is built, it is propagated to the intermediate container using the cross layer model described in Chapter 4. The instantiation is dual to the running "File saving" example (see Section 4.1.2), with `execve` being the $\texttt{SAVE}_S\texttt{()}$ event, the corresponding `exit` system call being the $\texttt{SAVE}_E\texttt{()}$ event and d_{str} being the data to be saved.

For the split events, the approach is similar: once `execve` of the split event is observed, the structured data stored in the intermediate container is retrieved and a number of cross actions (one per destination file) is initiated. Each cross action conceptually represents the split-event process action of "saving" the partial (according to the structure) data to each specific destination. In the case of split events, the cross action terminates upon observing the `CLOSE()` event on the destination file.

6.5.5. RQ1 - Precision

Metric

By design, the structured data flow tracking model is no less precise than the basic model described in Section 3.3.2, assuming the equivalence between a structured data item and the set of all the non-structured data item recursively stored in it (i.e. its flattened structure). It is therefore no surprise that after executing an arbitrary trace of events, the set of (flattened) data stored in a certain container c according to the structured tracking is always a (not-necessarily proper) subset of the data stored in c according to the basic model. Similarly, the estimation provided by the structured tracking is always a (not-necessarily proper) superset of the actual data stored in the container after executing the same trace, an indication of the soundness of the approach.

However, in order to quantify such improvement, the following metric has been used to measure precision:

$$precision(c) = \begin{cases} 1 & \text{if } \#actual(c) = \#estim(c) \\ \frac{\#actual(c)}{\#estim(c)} & \text{otherwise} \end{cases}$$

where

- $\#estim(c)$ is the number of different data items stored in a container c according to the considered data flow tracking model, and
- $\#actual(c)$ is the number of different data items *actually* stored in a container c. At the model level, this would be the output of an oracle. In the implementation, where c is a text file, such oracle returns the number of different lines in c. If c is an archive, it corresponds to the number of unique lines in all the text files stored (possibly recursively) in c, i.e. $actual(c) = \bigcup_{f \text{ archived in } c} actual(f)$

The set of unique lines in a document represents the different initial files the content of the document has been built from. Consequently, it corresponds to the *actual data* stored in the file. A sound data flow tracking must estimate *at least* such set of data for the respective document. A precise model would estimate *exactly* such set. If the model is too conservative, the estimation may also contain additional elements. Precision is measured in the number of such additional elements over the total.

Note that the computation of $estim(c)$ uses the flattened version of each data stored in c. Referencing the Excel example illustrated in Figure 6.3, this means that if the intermediate container c_1 contains the two non-structured data items $d1$ and $d2$ and the intermediate container c_2 contains d_{str}, which is a structured data item associated to the structure $Sheet3!B3 \rightarrow d1, Sheet4 \rightarrow d2$, then $estim(c_1) = estim(c_2) = \{d1, d2\}$ and $\#estim(c_1) = \#estim(c_2) = 2$, confirming the intuition that the precision does not improve for intermediate containers.

Results

As discussed in Section 6.5.3, different combinations of parameters have been tested during the generation of the event traces. It is worth noticing, first of all, that the two approaches model in a different way only events of class *zip* and *unzip*; the data flow propagation semantics ($\mathcal{R}$) for any other event is the same in the basic model and in the structured model. Also note that the precision in the modeling of *zip* events is the same for both models, as discussed at the end of the previous section.

Secondly, events of class *integrate* are the only ones that can introduce some imprecision in the structured model. To grasp the intuition behind this, consider the example of a file containing n different kinds of data. Intuitively, if only one line is read from this file, the tracking cannot distinguish which data has been read, and needs to assume that possibly all the n data have been read, a conservative overapproximation. Conversely, for the traces that do not contain *integrate* events, the structured model always provides maximum precision ($\forall c \in \mathcal{C} : precision(c) = 1$), while the basic model still introduces some overapproximation due to *unzip* events.

For these reasons, the tests in terms of precision focused specifically on these two classes of events, *integrate* and *unzip*.

Integrate events The first set of tests aimed at measuring the impact of the *integrate* events frequency over the precision. While fixing the probability for any other event class to the same value, different frequencies for the *integrate* events have been tested. Results

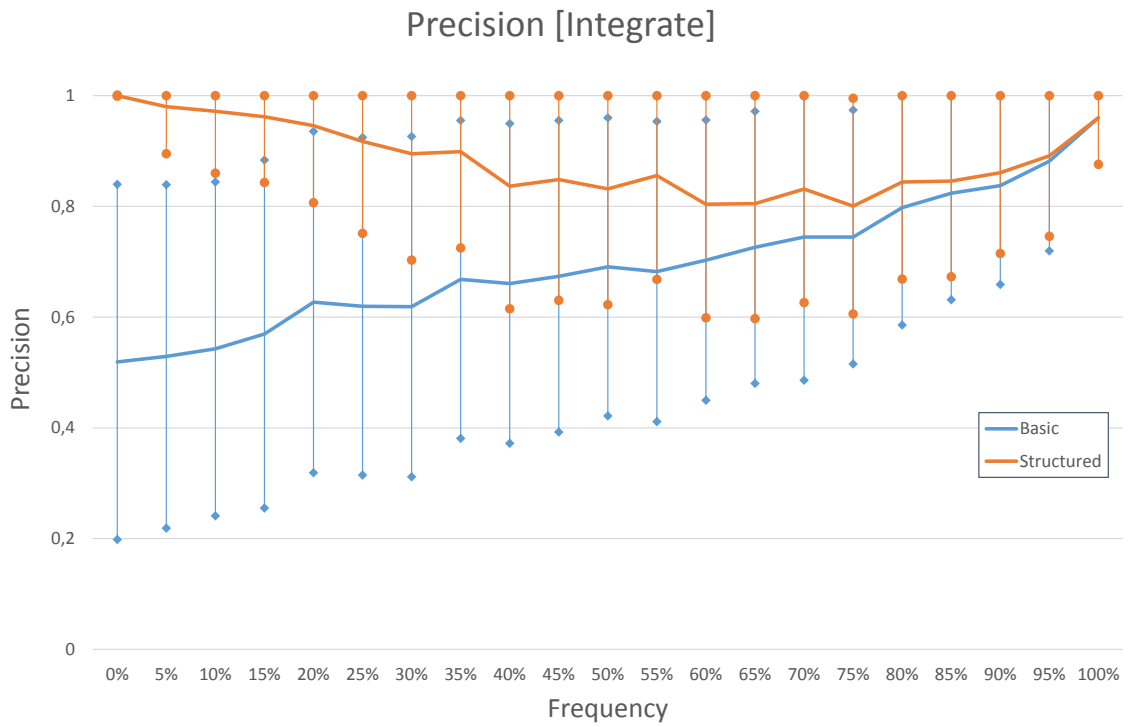


Figure 6.4.: Average precision for different frequencies of *integrate* events in the trace. All other events equally distributed among other classes. Traces composed by 250 commands. Standard deviation reported in error bars. Note that error bars top caps overlap for high frequencies.

are presented in Figure 6.4. The frequency of the *integrate* events is given on the x-axis. The y-axis represents the average precision for all the containers in the system at the end of the trace execution. Error bars indicate the standard deviation.

Figure 6.4 confirms the above intuition that *integrate* events have a negative impact on the overall precision of the structured data flow tracking. More precisely, Figure 6.4 shows that the higher the frequency of *integrate* events, the closer the basic model and the structured model behave. For frequencies of *integrate* events above 40%, the estimations provided by the two models are already statistically indistinguishable from each other.

It is important to note that in this set of experiments, each of the initial files contains 32K lines and that *integrate* events were configured to merge into the destination container 100 random lines from each of at most 5 different source files. Using smaller files or merging more content during *integrate* events would lead to a slightly different behavior: depending on the ratio between content being merged and size of the files, after a certain threshold the precision for both models starts to increase again with the frequency of *integrate* events, as Figure 6.4 shows for values above 75%. The intuition, in this case, is that the more often data is "mixed" among containers, the more likely it is that, at the end, all data have been transferred to the destination, making the estimation correct again. Recalling the example above of reading one line from a file containing n different data, the more often such event

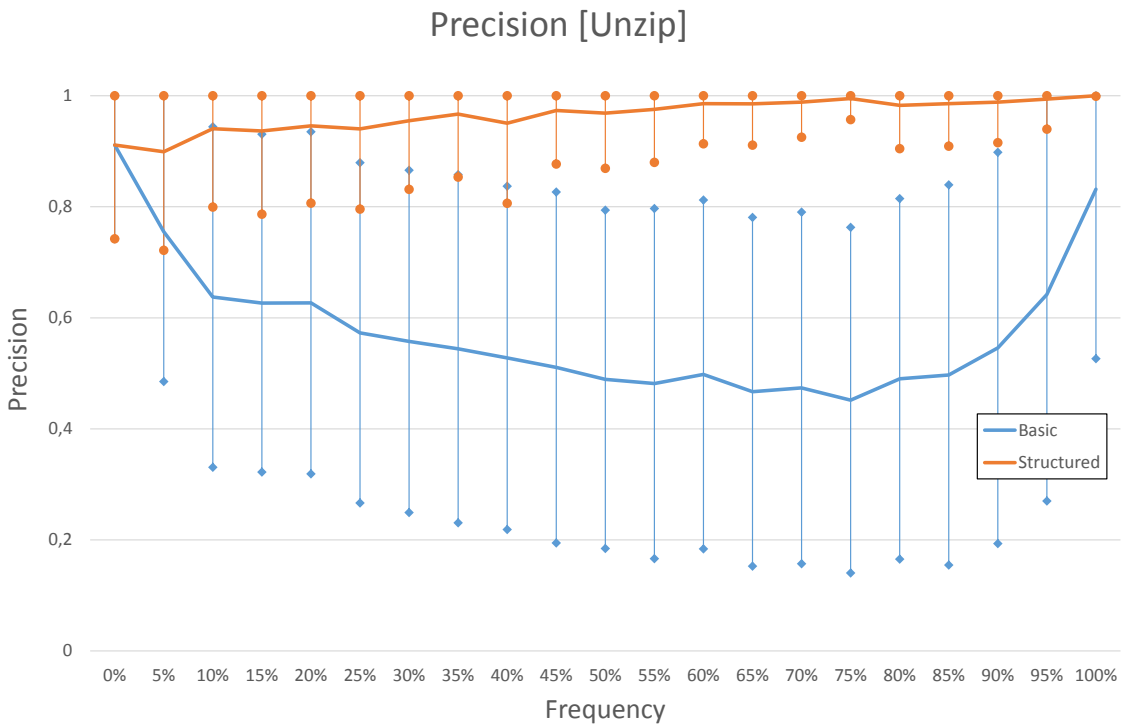


Figure 6.5.: Average precision for different frequencies of *integrate* events in the trace. All other events equally distributed among other classes. Traces composed by 250 commands. Standard deviation reported in error bars.

takes place among the same containers, the more likely is that, at the end, all the n data items end up being read, making the initial estimation correct.

Note that when the trace is composed only by *integrate* events, the two models provide identical estimations.

Unzip events The variable that mainly impacts precision in the structured tracking, however, is the frequency of *unzip* events. Similarly to the first one, the second set of experiments consisted in executing traces with different frequencies of *unzip* events. This is the only type of events for which the two approaches significantly diverge (the structured approach does not improve the precision for merge events, see Section 6.1.2); it is thus interesting to observe how much the estimation provided by the two models diverged for higher frequency of such events. As shown in Figure 6.5, the difference between the two graphs is significant, and while the structured model exhibits an asymptotic convergence to 1 (the higher the weight of the unzip events, the smaller the frequency of any other event, in particular of class *integrate*, the only ones that may introduce overapproximation), the negative impact of *unzip* events on the precision of the basic model is crisply clear.

Like in the previous set of experiments, for high frequencies of *unzip* events in the trace ($> 75\%$) the trend changes. In this case, the motivation is that a high frequency of *unzip*

events implies a low frequency of *zip* events, meaning that only few different archives are created. For this reason, the precision of the basic model starts to increase again, due to the fact that the overapproximation introduced by *unzip* events is limited to those few archives, i.e. the more *unzip* events in the trace, the less other files, for which the initial estimation is given and correct, are modified.

Note that traces with frequency of *unzip* events equal to 100% are actually composed by one *zip* event followed by equivalent *unzip* events (extracting the same files from the same archive), otherwise the traces would not be valid. In this extreme case, the precision after the trace is equivalent to the precision after executing a trace composed only by the first two events. Also note that for this set of experiments, the size of the initial files is irrelevant because every *zip*/*unzip* event reads and writes the whole content of the archive.

Similar kind of sensitivity tests were performed on other variables but none of them seems to directly affect precision as the frequency of *unzip* and *integrate* events does.

6.5.6. RQ2 - Performance

In order to answer **RQ2**, the third set of experiments compares the native execution time of the single traces to the time required to execute them with the basic monitor and with the monitor for structured data flow tracking. For this set of experiments, all event classes are assigned the same frequency, i.e. trace events are uniformly distributed among the different classes. The results are reported in Figure 6.6.

Compared to the native execution, the basic model is on average 4.3 times slower. Roughly half of the additional time is induced by the system call interception framework (PEP, see Section 5.1.1), while the other half is required for the modeling of the events (PDP, see Section 5.1.2, and PIP, see Section 5.1.3). The execution monitored with a structured model, in contrast, introduces an average overhead of 4.9x, out of which a factor of 0.2x is due to the computation of the checksum. In all the experiments, MD5 hash function has been used as checksum.

It is worth noting that the implementation of the PEP is taken from another work in the literature [91] and has not been optimized for the specific scenario considered. [91] is based on strace [9], which is a debugging userspace utility for Linux; other solutions, possibly based on different technologies, like virtual machine introspection [60], may offer better performance, but may also introduce additional issues, e.g. bridging the semantic gap [46]. Further solutions for runtime monitoring are described in related work (Chapter 9).

Note that the described implementation is intended to support *preventive* usage control; this means that the execution of every intercepted event is blocked until the PDP verifies that its execution is allowed. Performance could be significantly improved in case that *detective* enforcement suffices, by simply buffering event requests and processing them as a bulk from time to time, meanwhile directly allowing their execution.

Additionally, the implementation of the PDP and the communication interface between PEP and PDP are intentionally kept generic, in order to be used by different implementations at different layers of abstraction, like the Excel implementation described in Section 3.3.3; this includes, for instance, performing the communication between PEP and PDP using the Thrift protocol [56] over a local network connection, which is slower than a

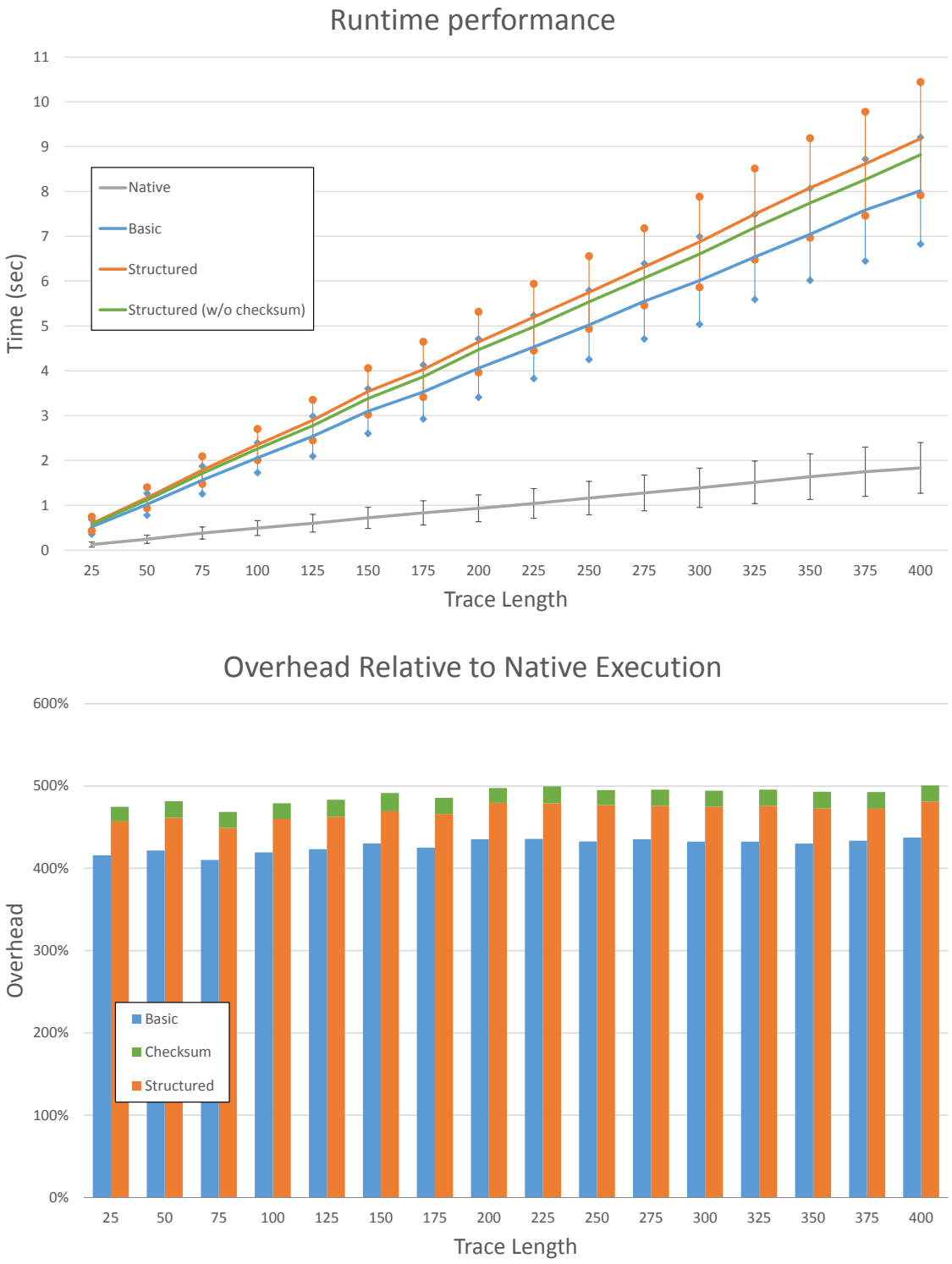


Figure 6.6.: Execution time comparison for traces of different length, absolute (top) and relative to the native execution time (bottom).

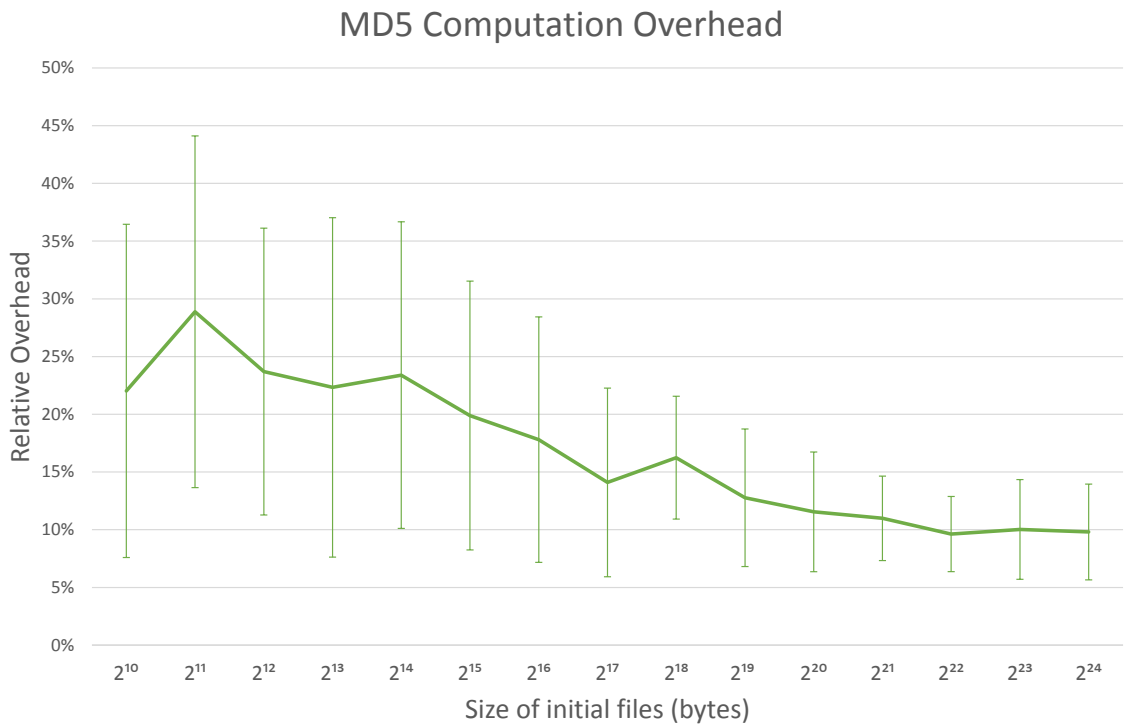


Figure 6.7.: Average time overhead introduced by the computation of the MD5 checksum for different sizes of the initial files. Standard deviation in error bars.

direct function call. The latter solution, however, offers less flexibility in terms of including additional monitors to the system.

Finally, the size of the files manipulated by the events significantly affects the native execution time of a trace and the time required to compute the checksum during each split and merge event. As shown in Figure 6.7, the bigger the size of the files, the smaller the impact of the checksum computation over the total execution time of the trace. In contrast, the time required for the modeling is independent of the size of the files. This depends on the fact that, in general, the time required to *model* an event is *independent* of the time required to *execute* an event. In other words, instantiations of the same model for layers of abstraction where events take longer, are less frequent and more complex than system calls, e.g. GUI interactions [100, 149, 105], are likely to result in smaller, when not negligible, relative performance overhead.

6.6. Challenges and Conclusions

This chapter presented a new idea for a generic model to reduce overapproximations in taint-based data flow tracking. It discusses a formalization of the model, and the implementation and evaluation of some exemplary instantiations. The key idea behind the presented model is to leverage information about events that propagate data in a structured manner. Section 6.5 formalizes a possible metric to quantify the precision of a data flow

tracking solution and evaluates an exemplary instantiation of the structured model w.r.t. an instantiation of the model presented in Chapter 3, both in terms of precision and performance. The evaluation identifies the different aspects of the system, like the computation of an MD5 hash or the type of a certain event, that influence such results. These results offer a specific answer to the fundamental research questions presented at the beginning of this chapter.

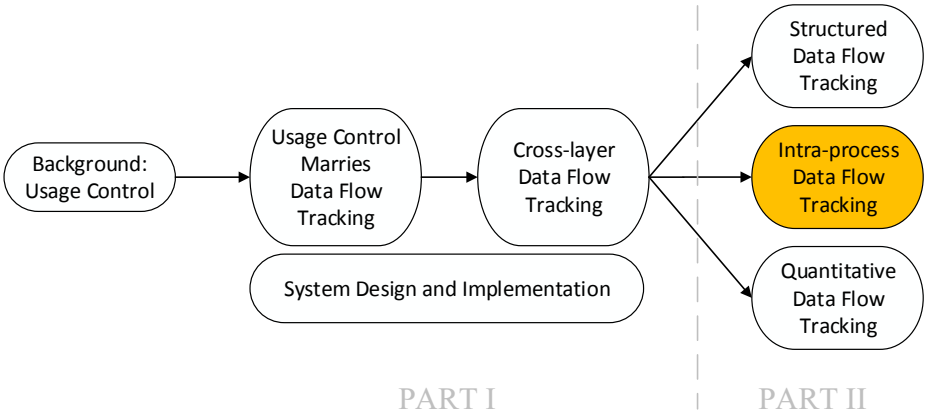
In terms of limitations, the lack of real world usage profile and the consequent fuzzy trace generation process pose an obvious threat to the validity of any results' generalization claim, as mentioned in Section 6.5.3. Additionally, this work assumes merge and split operations to be atomic. Yet, in some scenarios these operations actually correspond to sequences of events, like in the experiments described in the previous section. The approach suggested in Section 6.5.4, however, is not applicable to any scenario.

A more generic approach would be to model every event in the sequence using basic taint-propagation and then, in correspondence of the last event, replace the result with the structured taint mark. The drawback of this solution is that in some scenarios merge/split operations correspond to possibly infinite sequences of events, like a network stream of data. In these cases, split events may take place before the merge sequence is over. For this reason, an extension of the structured model that supports an incremental building of the structure is part of future work.

At a technical level, additional challenges come from finding appropriate $checksum$ and $partId$ functions for the concrete implementations, although a basic hash function like MD5 or SHA1 should work for most scenarios; the structured model is general enough to support any choice, as long as $partId$ maps each source container (before merge) and its respective destination container (after split) to the same partID.

7. Intra-process Data Flow Tracking

This chapter presents a second solution to improve the precision of the model introduced in Chapter 3. In this work, co-authored by the author of this dissertation and published in [107], the goal is achieved by leveraging static information flow analysis results.



7.1. Introduction

Information flow tracking [42, 43] tackles the problem of observing flows of data from sources (including input parameters to methods, sockets, files) to sinks (return values rendered on a screen, sockets, files). Information flow analysis systems can answer the question of whether or not data has (potentially) flowed or will (potentially) flow from a source to a sink. Different information flow analyzes cater for different definitions of source-sink dependencies, mainly distinguishing between *explicit* information flows (*direct* information flows or *data flows*) and control-flow dependencies, or *implicit* information flows. Data flow tracking solutions are generally tailored to one particular layer of abstraction, like source code, byte code, machine code, or the operating system layer (see Chapter 9).

As described in the first parts of this dissertation, data flow tracking technologies can be used to support distributed data usage control concepts. Chapter 4 discusses in detail the importance of performing data flow tracking at *multiple layers of abstraction*, to the end of expressing more complex system-wide policies such as "any representation of my data must be deleted after thirty days" and to preserve the *high-level semantics* of objects (e.g. "a mail") and events (e.g. "forward"), which is otherwise hard to capture at lower layers. But this benefit comes at a price: even a small number of monitors running in parallel may seriously compromise the performance of the overall system, and dedicated high-level monitors are not always available for every domain. In these cases, the usual solution is to rely on conservative estimations provided by lower layers.

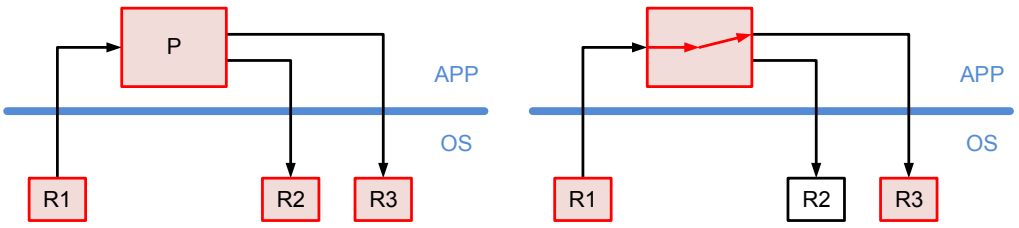


Figure 7.1.: Example of data flow tracking without (left) and with (right) information about how data flows through the application. Note that in the second case, no flow of data towards resource R2 is reported

For instance, as depicted in Figure 7.1, if a dedicated monitor for a process is not available (see Figure 7.1, left), an OS-layer monitor would have to treat the process as a "black box" and assume that every sensitive data it got in touch with (resource R1) is propagated to every future output. This solution likely introduces many false positives and in this sense grossly overapproximates the set of potential information flows. Knowing that only certain outputs depend on a particular input (see Figure 7.1, right), the analysis can be propagate the sensitivity of the inputs only to those outputs and not to the others.

This chapter proposes an approach to retrieve and leverage this information to mitigate the overapproximation issue. The core idea behind this solution is to replace the runtime monitoring of how data flows through a process (or its *black-box* overapproximation) by consultations of a *statically pre-computed mapping* between its inputs and outputs.

The fundamental research questions addressed in this chapter are

How can the precision of the data flow tracking model be augmented with static information flow analysis results?

How can the improvement in terms of precision be quantified?

Which aspects of the static analysis affect the precision of the approach and how?

What is the overhead induced by this approach and what influences it?

7.1.1. Example Scenario

> *Alice works as analyst for ACME Inc., a smartphone manufacturer. In order to prevent employees to work with out-of-date information, ACME enforces the policy* "upon logout, delete every local copy of reports or contacts data". *After every login, Alice has to download from the central repository the fresh version of the reports she is interested in and the address of the suppliers or customers she needs to send the reports to. In this context, a* ***data flow tracking*** *system is in place to automatically find every* ***copy of sensitive data to be deleted*** *within the system.*

> *However, if the* ***tracking is imprecise*** *and introduces many false positives, additional possibly* ***important resources may be accidentally deleted*** *as well.*

Consider the setting illustrated in Figure 7.2: Alice uses the *Zipper* application to archive multiple reports into a single archive file, which she then sends to the company's Ftp-Server using the *Ftp-Client* application. In this scenario, an archive (`File3`) is created from different reports (**E** and **F**) and sent to a remote server. Assume that data **E** and **F** are protected by the "delete upon logout" policy, which is enforced by a usage control infrastructure at the OS-layer. Current usage control solutions for operating systems [164, 73] consider each running process as a black box, i.e. they do not track how data flows through the application, but limit their observation to the interactions of the application with the OS. When sensitive data **E** and **F** are read by the Zipper application (Zipper's `Source1` in Figure 7.2) all further output may contain (possibly partially) **E** and **F**, and are therefore subject to the "delete upon logout" constraint.

This includes, among the rest, `Sink2`, used to automatically store Zipper's updated configuration settings into file `ZipConfig` upon exit. If this happens, then Zipper's configuration file will be deleted upon logout, compromising the functionality of the application. Similarly, every other file created by Zipper afterward would be marked as possibly containing **E** and **F**, and thus deleted.

The same concerns also apply for the Ftp-Client. FTP works with two channels, one for commands, and one for payload. In a black-box monitoring situation, once sensitive data is read, every write to any of the two channels may be possibly carrying sensitive information, and, as such, it should propagate the taint to the socket connection, and possibly to the recipient side. In this case, the credentials on the Ftp-Server side sent via the command channel would also be marked as "to-be-deleted".

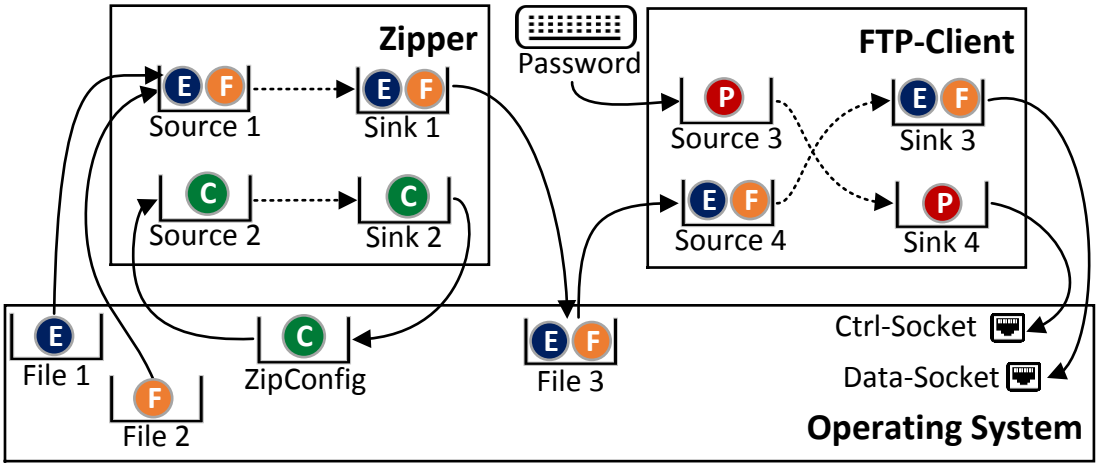


Figure 7.2.: Example inter-application data flow [107]. Data **E**+**F** enter Zipper from `Source1` and leave via `Sink1`, enter Ftp-Client via `Source4` and leave via `Sink3`.

7.1.2. Summary

The approach presented in this chapter, detailed in Section 7.2, improves the precision of information-flow tracking *system-wide*, i.e. through and in-between different processes/applications, like the flow of data **E** and **F** through the Zipper application (`Source1`→ `Sink1`) into `File3` and then through the Ftp-Client application (`Source4` → `Sink3`) till the payload channel. Section 7.3 shows that this solution introduces lower execution overhead than other dynamic monitors for comparable scenarios.

Problem. The concurrent execution of multiple monitors at different layers of abstraction allows to exploit high-level semantic information (e.g., "screenshot" or "mail") but is performance-wise expensive and requires dedicated monitoring technologies for every layer/application. On the other hand, relying only on estimates provided by other layers (e.g., the black-box approach) improves performance, but at the price of (possibly significant) precision loss.

Solution. This chapter proposes a dynamic monitoring approach for generic processes that replaces runtime intra-process data flow tracking by consultations of a statically computed taint-propagation table. Such a monitor is more performant than equivalent runtime monitors for the same application and more precise than the OS-layer overapproximation adopted when such a monitor is not available.

Contribution. This work represents the first combination of static and dynamic data flow tracking for different layers of abstraction and *through multiple different applications*. This solution improves the precision of OS-layer data flow tracking with minimal intra-process runtime tracking overhead.

7.2. Approach

The work described in this chapter considers a setting with monitors at two layers of abstraction: a dynamic monitor at the OS-layer, based on system-call interposition [73] like the one described in Section 3.3, and an inline reference monitor at the application level, which observes the execution of source and sink events. The goal is to improve tracking precision at the OS-layer with minimal performance penalties. Although the approach is generic in nature and could be applied to any language or binary code, this work focuses on an instantiation for the Java Bytecode (JBC) layer.

In the following, the standard terms from the literature *source* and *sink* will be used to indicate, respectively, a method (or one of its parameters) invoked by the application to input data from the environment, and for the dual invocation to output something to the environment. In special contexts, like Android, it is possible to find in the literature detailed lists of source and sink methods [140], but in general the choice is left to the analyst. In this work, a source (sink) is the invocation of a Java standard library method that overrides any overloaded version of InputStream.read (OutputStream.write) or Reader.read (Writer.write), or a method that indirectly invokes one of them, e.g., Properties.load(), which uses an input stream parameter to fill a properties table.

The monitoring approach in this case roughly consists of tracking flows of data through an application by replacing the runtime monitoring of each event with the static mapping between its inputs and its outputs: if a source in an application is executed, the respective

input's taint mark is stored. When a sink is executed, all sources (and therefore all taint marks) with potential flows to this sink are determined using a static mapping of potential flows between sources and sinks. There is hence a need to instrument sources and sinks, but *not all the instructions in-between them*.

The approach consists of the following three phases:

A. Static analysis In this phase an application X is analyzed for possible information flows between sources and sinks. The result of this phase is a report containing a list of all sources and sinks in the application and a mapping from each sink to all the sources it may depend on.

B. Instrumentation In this phase all sinks and sources identified by the static analysis (and *only* those instructions) are instrumented in the bytecode of X, allowing to monitor their execution.

C. Runtime In this phase the instrumented application is executed. Every time a source or a sink of X is executed, the injected monitor interacts with the monitor for the OS to exchange information about the data being read or written using the cross-layer data flow tracking approach described in Chapter 4.

In the remainder of this section, these phases will be described in detail, using Zipper and Ftp-Client as examples. Notice, however, that in principle the same approach can be applied in a push-button fashion to any Java program.

7.2.1. Static Analysis

In the static analysis phase, the code of a given application is statically analyzed for information flows, in order to create a report containing the list of all sources and sinks in the code and the dependencies between them.

This goal is achieved using JOANA [6, 70], a state-of-the art static information flow analysis tool for Java applications, built on top of the WALA framework [10]. Notice that this generic approach is not bound to any specific tool, but it can leverage any static information flow tool. Also note that the techniques used by JOANA are also used by other static analysis tools (e.g. [153]).

JOANA operates in two steps: first, it builds a *Program Dependence Graph* (PDG) [54] of the application; second, it applies slicing-based information flow analysis [72] on the PDG to find out which set of the sources influences which sinks. In order to reduce the number of false positives, JOANA leverages several program analysis techniques. The remaining of this subsection elaborates on some fundamentals of the technologies used by JOANA, namely PDGs, slicing, and some of the various analysis options offered by the tool.

PDGs and Slicing. A PDG is a language-independent representation of a program. The nodes of a PDG represent statements and expressions, while edges model the syntactic dependencies between them.

There exist different kinds of dependencies, among which the most important are *data dependencies* and *control dependencies*. Data dependencies occur whenever a statement uses a value produced by another statement, whereas control dependencies arise when a statement or expression controls whether another statement is executed or not. The PDG in Figure 7.3 contains a data dependency between the statements in line 1 and in line 2 because the latter uses the value of x produced by the former. It also contains a control dependency

```
1  x = 1;
2  y = 2x - 5;
3  if (y > 42) {
4    z = 1;
5   } else {
6    z = 2;
7   }
```

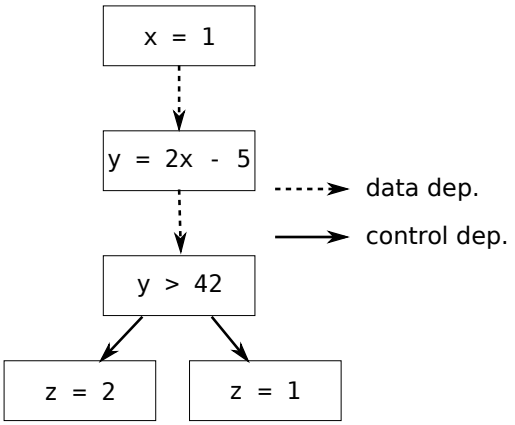


Figure 7.3.: A code snippet and its PDG [107].

between the if-statement in line 3 and the statements in line 4 and line 6 because whether line 4 or line 6 is executed depends on the value of the if-expression in line 3.

PDG-based information flow analysis uses *context-sensitive slicing* [142], a special form of graph reachability: given a node n of the PDG, the *backwards slice* of that node contains all nodes from which that node is reachable by a path in the PDG that respects calling-contexts. For sequential programs, it has been shown [84, 162] that a node m not contained in the backwards slice of n cannot influence n, hence PDG-based slicing on sequential programs guarantees *non-interference* [68]. It is also possible to construct [95, 64] and slice [125, 65] PDGs for concurrent programs. However, in concurrent programs there are additional kinds of information flows, e.g. probabilistic channels [147], so that mere slicing is not enough to cover *all* possible information flows between a source and a sink. A PDG- and slicing-based algorithm providing such guarantee has recently been developed and integrated into JOANA [64, 66].

Analysis Options. JOANA is highly configurable and allows to configure different aspects of the analysis, e.g. to ignore all control flow dependencies caused by exceptions, or to specify the different types of *points-to analysis* [17] that are used to build the PDG. Points-to-analysis is an analysis technique which aims to answer the question which heap locations a given pointer variable may reference. JOANA uses points-to information during PDG construction to determine possible information flows through the heap and therefore depends heavily on the points-to analysis precision.

There exist several kinds of points-to analyzes which lead to a variety of concrete analyzes [146] of different precisions and costs. One dimension of precision is *context-sensitivity*. Intuitively, a context-sensitive points-to analysis distinguishes multiple instances of methods by some property of their calling contexts. This keeps the points-to sets for reference variables which are local to the method (like parameters and local variables) separated for distinguished calling contexts and hence allows for more precision in subsequent analyzes.

JOANA supports different flavors of context-sensitivity. This work uses only two of them, k-CFA and *object-sensitive* points-to analysis. *k-CFA* [150, 117] uses as context information the top k items of the runtime call stack, i.e. the history of the last k call sites (i.e.

methods and bytecode offsets of the particular call instruction) that eventually transferred control to the method under analysis. In contrast, *object-sensitive* [118] points-to analysis does not consider the runtime call stack, but instead uses the *receiver object* as discriminator, i.e. it distinguishes multiple instances of a non-static method by the object on which the method was called.

More details on the different points-to analyzes, which are not the main scope of this dissertation, can be found in the cited related works. Note that, once again, the general idea behind this approach is independent of the specific tool (e.g. JOANA) or the specific points-to analysis adopted.

The outcome of this phase is a list of the sources and sinks in the code of the application and a table that lists all the sources each sink depends on.

7.2.2. Instrumentation

In this phase, the report generated by the static analysis is used to instrument each identified source and sink. For each source or sink, the analysis reports the *signature* and the exact *location* (parent method and bytecode offset).

```
1  void zipIt(String file, String srcFolder) {
2   fos = new FileOutputStream(file);
3   zos = new ZipOutputStream(fos);
4   fileList = this.generateFileList(srcFolder);
5   byte[] buffer = new byte[1024];
6   for (String file : fileList) {
7    ze = new ZipEntry(file);
8    zos.putNextEntry(ze);
9    in = new FileInputStream(file);
10   int len;
11   while ((len = in.read(buffer)) > 0)
12    zos.write(buffer, 0, len);
13  in.close();}}
```

Listing 7.1: Java code fragment from Zipper

Consider the code snippet in Listing 7.1, used in the Zipper application: static information flow analysis detects the flow from the source at line 11 (`Source1`), where the to-be-zipped files are read, to the sink at line 12 (`Sink1`), where they are written into the archive. Listing 7.2 shows the corresponding analysis report: line 1 to line 9 specify that the return value of the `read` method invocation at bytecode offset `191` in method `zipIt` is identified as `Source1`. The same holds for `Sink1` (line 11 - line 19), but in this case the first parameter (line 18) is a sink, not a source. In the final part, the report also provides information about the dependency between `Sink1` and `Source1` (line 21 - line 25), which is then used to model possible flows.

Each reported source and sink is wrapped with additional, injected bytecode instructions using the *OW2-ASM* [8] instrumentation tool. In the following, the set of these additional instructions is referred to as *inline reference monitor*. The outcome of this phase is an instrumented version of the original application, augmented with a minimal inline reference monitor that interacts with the OS-layer monitor when a source or a sink is executed. This way incoming/outgoing flows of data from/to a resource, like files or network sockets, can be properly modeled.

In terms of the architecture described in Chapter 5 and of the $\dot{\mathcal{R}}_{A\otimes B}$ algorithm for cross-layer data flow tracking described in Chapter 4, the reference monitor injected in this phase

acts as a PEP that notifies the PIP component about the beginning and the end of the execution of a source/sink bytecode instruction; these events can be seen as the *start* and *end* events at the application level in the Excel examples described in Section 3.3. In this sense, the execution of a source (resp. sink) instruction is modeled like the `LOAD()` (resp. `SAVE()`) events in Excel.

```
<source>
  <id>Source1</id>
  <location>JZip.zipIt(Ljava/lang/String;Ljava/lang/String;)V:191
  </location>
  <signature>
    java.io.FileInputStream.read([B)I
  </signature>
  <return/>
</source>
...
<sink>
  <id>Sink1</id>
  <location>JZip.zipIt(Ljava/lang/String;Ljava/lang/String;)V:185
  </location>
  <signature>
    java.util.zip.ZipOutputStream.write([BII)V
  </signature>
  <param index="1"/>
</sink>
...
<flows>
  <sink id="Sink1">
    <source id="Source1"/>
  </sink>
</flows>
```

Listing 7.2: Static analysis report listing sinks, sources and their dependencies

7.2.3. Runtime

This phase of the approach corresponds to the actual runtime data flow tracking, where the instrumented application is executed in a dynamically monitored OS. At runtime, a single OS-layer monitor exchanges information with multiple inlined bytecode-level reference monitors, one per application. Note that by assumption, the information to be tracked is initially stored somewhere in the system, e.g. in some files or coming from certain network sockets, and marked as sensitive. For instance, in the example of Figure 7.2, data **E** and **F** are already stored in `File1` and `File2`, respectively.

Once a source instruction is about to be executed, the instrumented code queries the OS-monitor to obtain information about the sensitivity of the content in input. It then stores this information associated to the source id (e.g. `ZipConfig` $\rightarrow$ `Source2` in Figure 7.2). When a sink instruction is about to be executed, the instrumented code fetches the information about the sensitiveness of *all the sources the current sink depends on* according to the flow map in the analysis report (`Source2` $\rightarrow$ `Sink2`). Such information denotes all the possible inputs the current output may depend on, but, most importantly, it denotes all the inputs the current output does *not* depend on: this is where the overapproximation is mitigated.

With this approach, even if the application reads additional data (like data **E** in Figure 7.2) before generating the output, the sensitivity associated to the sink (and, conse-

quently, to the output file) remains the same, as long as the additional input is not used to generate the output. In contrast, if the process is treated as a black-box, then every output will be as sensitive as the union of all the sources acquired till then.

Lastly, the information about the content being output by the current sink (`Sink2` $\rightarrow$ `ZipConfig`) is forwarded back to the OS monitor, so it can carry on the tracking outside the boundaries of the single application.

7.3. Evaluation

The goal of the work described in this chapter is to improve data flow tracking precision at the OS-layer, possibly introducing minimal runtime overhead. For this reason, the work is evaluated in terms of *precision* (false positives[1]) and in terms of *performance*. The evaluation addresses the following research questions by means of case studies:

RQ1 How much more precise is this approach with respect to the estimation provided by the OS-layer monitor alone?

RQ2 How long does the static analysis phase take?

RQ3 How much slower is the instrumented application, and how does it compare with a purely dynamic solution?

Firstly, the analysis is performed on the applications described in the running example, Zipper and Ftp-Client. The Zipper application ($\sim$400 lines of code) has been developed for the purpose of this evaluation, while the Ftp-Client was found online [5]. The code of these apps is intentionally minimal, in order to facilitate manual inspection of the results. Moreover, these applications stress-test the proposed solution because the approach instruments only entry and exit points in the code (sources and sinks), but the vast majority of executed instructions are indeed sources or sinks; for comparison, the same solution has also been executed on a third use case, an application with little I/O and large amount of computation in-between: the Java Grande Forum Benchmark Suite [4], a benchmark for computationally expensive Java applications. Among others, this framework has been chosen to compare the evaluation results to those of related work [31].

7.3.1. Settings

The evaluation was performed on a system with a 2.6 GHz Xeon-E5 CPU and 3GB of RAM. All the applications have been a statically analyzed using the different configurations described in Section 7.3.2. For each of these configurations, the median execution time of at least 30 independent executions is reported, to weed out possible environmental noise. The implementation used as OS monitor, has been already presented in Section 3.3.2.

All the runtime experiments use the objsens-D (see Section 7.3.2) report from the static analysis phase. The decision is motivated by the highest precision of this configuration in preliminary tests; however, tests with any other analysis resulted in statistically indistinguishable runtime performances.

[1] Static analysis is assumed to be correct, i.e. all the real flows that can be statically identified are correctly reported by the analysis by assumption, without any false negative. Limitations of this approach are discussed in Section 7.3.2

7.3.2. RQ1 - Precision

To answer question **RQ1** some considerations need to me made: First, by construction, the approach cannot be less precise than treating the processes as black boxes (= every output contains every source read so far), the typical conservative estimation made by monitors based on system-call interposition [73]. Second, while dynamic analyzes usually rely on explicit flows only, static analyzes may consider additional kinds of dependencies between instructions (e.g. control-flow dependencies), generating more dependencies between sources and sinks. Third, even when configured to report explicit-flows only, a static approach considers *every* possible execution at once, meaning that if at least one execution leads to the flow, then the sink statically depends on the source.

```
1 in=input();
2 if (cond) {
3   out=in;
4 }
5 output(out);
```

For instance, for the simple sequence of instructions on the left, the analysis will report that the sink at line 5 depends on the source at line 1, although this is true only for those values that make condition `cond` at line 2 evaluate to true. A runtime monitor would report the dependency only during those runs where `cond` is satisfied. Replacing the runtime monitoring with a static dependency table introduces overapproximation by making the sink depending on the source during *every* execution, regardless of `cond`'s value.

To perform experiments on the scenario described in Section 7.1.1, three files have been created, two filled with random content and assigned to data **E** and **F** respectively, and one containing the configuration for Zipper, assigned to data **C**. In this scenario, the only data read from the standard input is the password, thus marked as **P**. Then, after executing the scenario (i.e. archiving the files using Zipper and sending them to the server using Ftp-Client), the different monitors must estimate the sensitivity of the content that reached the ftp sockets at the end of the run. As expected, executing the scenario in the presence of just a monitor for the OS-layer (*black-box* approach), the estimation is rather coarse-grained (every data reached both sockets). The execution with the proposed solution, instead, provided the expected result (data **E** and **F** flowed only to the data socket, while **P** flowed only to the control socket).

This approach is by construction no less precise than the black-box approach, so it is no surprise that this result confirms the improvement in terms of precision of the solution. However, it is hard, if not impossible, to *quantify* such improvement in general terms, in order to answer **RQ1**. Considering that a black-box approach would always be as precise as this approach when every source is connected to every sink, one possible metric for precision improvement could be based on the number of source-to-sink dependencies that can be safely discarded by the static analysis. With this argument, let $\#flows$ denote the number of statically computed dependencies between sources and sinks. Precision is then measured as

$$1 - \frac{\#flows}{\#sources \times \#sinks}$$

where 0 indicates that every source flows to every sink (like in the black box approach) and 1 indicates that all the sinks are independent from the sources, i.e. no data propagation. Literature offers no better metric to measure precision of static analysis with respect to black-box dynamic monitoring.

The static analyzer is run on both the Zipper and the Ftp-Client with different analysis options (see Section 7.2.1). As depicted in Table 7.1, various *points-to analyzes* have been

	Points-To	Time (s)	#Sources/#Sinks	Precision (DI / D)
Ftp-Client	0-1-CFA	32	9 / 46	38% / 51%
	1-CFA	64	9 / 46	58% / 73%
	2-CFA	153	9 / 46	58% / 73%
	objsens	220	9 / 46	38% / 74%
Zipper	0-1-CFA	53	10 / 56	24% / 43%
	1-CFA	82	10 / 55	25% / 53%
	2-CFA	185	10 / 55	55% / 78%
	objsens	353	10 / 55	57% / 84%
JGFBS	0-1-CFA	211	8 / 84	56% / 59%
	1-CFA	580	8 / 81	71% / 75%
	2-CFA	626	8 / 81	71% / 77%
	objsens	360	8 / 81	73% / 79%

Table 7.1.: Static analysis results for different configurations [107]. Precision is defined as $1 - (\#flows/(\#sources \times \#sinks))$
.

tested (0-1-CFA [71], 1-CFA, 2-CFA, object-sensitive) with (DI) and without (D) considering implicit information flows. According to the formula above, the improved precision of the instrumented version of the applications varies between 24% and 84% for the Zipper, between 38% and 74% for the Ftp-Client and between 56% and 79% for the Java Grande Forum Benchmark Suite, depending on the configuration. Although object-sensitive points-to analysis and k-CFA are incomparable in theory, object-sensitivity tends to deliver more precision in various client analyzes [118, 104]. This effect can in part also be observed here, at least for the configurations in which indirect flows are ignored (D). The reason is that the points-to analysis result is mainly used to compute the data dependencies and has only limited effect on the control dependencies. For the Ftp-Client, object-sensitivity delivered worse precision than 1- and 2-CFA.

Note that these numbers are hard to relate to dynamic values, because they depend on the specific application under analysis and they do not take into account how many times a certain source/sink instruction is executed at runtime.

7.3.3. RQ2 - Performance of the Static Analyzer

The results of the evaluation of the static analyzer are shown in Table 7.1. The constructed PDGs have between 7×10^4 and 5×10^5 nodes and between 6×10^5 and 9×10^6 edges. In the "direct-flows-only" (D) configurations, PDGs have between 11% and 34% fewer edges than in the respective "direct-and-indirect-flows" (DI) configurations. In general, most of the execution time (80-90%) was spent on PDG construction, whereas the majority of the remaining time was spent on slicing. The execution time of the static analyzer largely depends on the points-to analysis used to build the PDG. This comes from the fact that the actual PDG construction makes heavy use of the result of the points-to analysis, which can greatly vary in size with different settings: a larger points-to result makes the PDG construction phase more expensive.

	Size (bytecode) orig.→instr.	Average overhead per sink/source		Overhead in total				
		Intra	Intra+OS	Intra	Intra+OS	[93]	[31]	[94]
Zipper	1611 → 2192	2.06x	22.92x	2.09x	34.28x	220.4x	-	-
Ftp-Client	9191 → 9785	0.16x	4.37x	0.28x	6.75x	25.7x	-	-
JGFBS	29003 → 30123	6.33x	144.65x	0.001x	0.07x	10.5x	0.25x - 1x	-
$Zipper_{32}$	1611 → 2192	0.24x	7.11x	0.33x	11.61x	19.7x	-	*15.2x - 28.1x*

Table 7.2.: Runtime analysis results. Underlined value taken from the literature, all others measured. Values in italic refer to results of comparable tests (see Section 7.3.4). $Zipper_{32}$ indicates the archiving of 261MB using internal buffers 32 times bigger.

7.3.4. RQ3 - Runtime Performance

The proposed solution has been compared to other information-flow tracking solutions from the literature, either by running the tool on the same scenario used to evaluate the solution from the literature [31] or, when possible, by running the tools from the literature on the same test cases chosen for this work. The latter is the case for LibDFT [93], a tool for binary-level instrumentation based on the Intel PIN [113] framework, used to track flows of data through a certain process. [31] instead is a hybrid information-flow tracking tool for the Java virtual machine. Other candidate tools for the comparison (like [36, 41, 175]) could not be executed, because their source codes were tailored to specific legacy systems, libraries or environments, or lacked the necessary documentation to configure them properly and to get them running for comparison purposes.

Time measurements are reported in Table 7.2, published in [107]. The first column shows the difference in size between the original application (including libraries, if different from the standard Java APIs), measured in number of bytecode instructions.

The evaluation of runtime performance is based on multiple experiments: recalling the scenario in Section 7.1.1, the first set of experiments simulates the transfer of a 20K file to a remote server using Ftp-Client and its compression using Zipper. Another set of experiments consisted in running the tool on a computationally expensive task with limited I/O, the Java Grande F.B.S., used in [31]'s evaluation. The results consist of the total time required to execute each test case 1) natively (i.e. without instrumentation), 2) with just the intra-process monitor but without the monitor at the OS-layer (columns Intra in Table 7.2), and 3) with monitors at both layers (columns Intra+OS).

As mentioned before, the Zipper and Ftp-Client applications are stress-testing the approach because they transfer data in blocks of 1KB at a time. This results in a huge number of read/write events: for comparison, creating a zip file of 261MB of Linux source code with Zipper generated ∼122K write and ∼256K read events, whereas *gzip*, an equivalent tool used in [94]'s evaluation, only generates 3781 writes and 7969 read system calls for the same input and the same output.

Because [94] is a dynamic monitor that connects information flow tracking results for multiple applications on multiple hosts in the same system (system-wide tracking), a comparison to this work is particularly relevant. To perform it, an additional set of experiment has been executed: compressing 261MB with the Zipper application after increasing the

size of the buffers by a factor of 32x. This way, Zipper generates the same number of I/O events of the tool used in [94] when executed on the same inputs. Although comparing different applications (Zipper and gzip) is always tricky, since the number and type of generated events is almost identical, the comparison is still informative and likely fair. These results are presented in the last row of Table 7.2. The overhead for zipping 261MB (11.61x) using Zipper is smaller than the best value for gzip mentioned in [94] (15.2x-28.1x). Similarly, the overhead on the Java Grande F.B.S. (0.07x) is one order of magnitude smaller than the results in [31] (0.25x-0.5x).

Notice that the static analysis and the instrumentation phases take place only once per application. For this reason, they are excluded from the computation of the relative runtime overhead (columns Intra and Intra+OS in Overhead, Table 7.2). Also, the values in Table 7.2 do not include the time required to boot the Java Virtual Machine, which is independent of the instrumentation and thus irrelevant. It is worth noticing that while different configurations of LibDFT has been tested, in the best case only overheads one order of magnitude larger than those reported in the original paper[93] could be reproduced.

7.4. Discussion

This section elaborates on some of the technical and fundamental highlight and limitations of this approach and provides a general summary of the experimental results.

By combining static and dynamic data flow technologies, the approach described here managed to analyze and to track system-wide information flows between different applications and across different application layers (and different systems). As results show (see Table 7.1), depending on the applied *points-to* analysis it is possible to tune the precision of the tracking system significantly. Although the runtime overhead highly depends on the number of instrumented sources/sinks and on how an application is implemented, in the aforementioned case studies (a non-optimized implementation of) this approach can perform better than existing solutions in terms of either, precision or performance.

On computationally intensive tasks, for instance, like the Java Grande F.B.S., the tool exhibited a negligible overhead in practice ($<$0.07x), while in other I/O intensive scenarios, the performance is comparable to, if not worse than, existing solutions from the literature. To better understand the factors that influence the performance, the remainder of this section analyzes some of the limitations of the general approach described here.

Firstly, static analysis is by its nature not able to distinguish every possible execution and therefore introduces overapproximations, which results in imprecision in the information flow analysis. One possibility to improve precision, is the use of more precise points-to analyzes (see Section 7.2.1), but this usually comes at the price of considerably longer analysis times (see Table 7.1) and higher memory consumption, meaning scalability problems.

This scalability issue can be mitigated by tuning the implementation. Also, the static analysis only has to be performed once per application and could be outsourced to a high performance server which precomputes the tables once for each program. It is also reasonable to assume that in a real scenario the source-sink dependency table comes together with the application in a signed form, similar to proof-carrying code. This obviously introduces some key management issues, which are out of the scope of this work and therefore not discussed.

The scalability problems are worsened by the fact that even small Java programs use large parts of the Java standard library – sometimes just referencing a prominent class name causes a whole avalanche of static initializers to be executed –, which makes the structures which JOANA constructs (call graph, PDG) very large. Currently, JOANA performs a *whole-program analysis*, which means that all the libraries used by the code under analysis need to be analyzed every time. There exists an approach to make the PDG construction more modular by pre-computing appropriate approximations of library PDGs and re-using them when calls of library methods are encountered [69], but this approach has not been fully integrated yet, so it is unclear whether it brings considerable performance gains in practice.

Another limitation of JOANA (and of static analysis in general) is the inability to properly analyze applications that do not have a main entry point but that are used through callback handlers (e.g. Swing): Analyzing callback-based applications requires a model that captures the way the callback handlers are used (e.g. which simulates the user). Such a model could for example be obtained by running the application, by specification in a dedicated language or by simulating all possible callback connections.

Also, JOANA currently ignores reflective code. Like callbacks, dealing with reflection in a sound but not overly imprecise way is not a JOANA-specific issue, but rather a fundamental challenge in static analysis, for which a general precise solution is impossible. Additional analyzes like string analysis may help to resolve some reflective code (e.g. find out the name of a dynamically loaded class) but in general, either very coarse assumptions have to be made, or unresolvable reflective code has to be ignored. Some approaches like [29] exploit runtime information to resolve reflection.

In the context of this work, it is important to clarify the notion of information flow used for the static analysis phase. In the examples described here, for instance, information flows solely caused by exceptions have been intentionally ignored. This has to do with the fact that every I/O operation may cause an exception, making the execution of every source influencing every following sink by possible failing. While the proposed tool can handle exceptional control-flow, this feature has been disabled in the tests.

Note that if the static analysis phase is configured to detect explicit flows only, then the combination of the runtime monitors for the OS and the inlined reference monitor in the application guarantees a property similar to the one described in Section 4.2.1, based on Volpano's *weak secrecy* [158]. However, it would be easy to circumvent the analysis by transforming each direct assignment within the application into an "indirect" assignment (i.e. a loop that leaks the value of a variable one bit at a time via a control-flow dependency). This way, the analysis would report no dependencies between sources and sinks. On the other hand, sound and precise system-wide non-interference assessments (including implicit flows) require a static analysis of *all* the applications together at once. This is because independent analyzes for the single applications are inherently non-compositional, e.g. they cannot model dependencies generated by the concurrent interactions on shared resources [147]. Due to its exponential nature, a global all-at-once analysis would be unfeasible even for a small number of applications of a reasonable size and would also likely lead to results that are too conservative to be useful (i.e. too many false positives).

The approach described in this chapter lies somehow in-between these two extremes: by considering implicit flows during the intra-process tracking, for each application it can

guarantee the non-interference between inputs and outputs if they do not appear in the report, while at the same time it also models the flow of data across different applications. This property is stronger than weak secrecy and than the single-layer soundness (see Section 3.4), which ignores intra-process implicit flows, but still weaker than system-wide non-interference, due to the aforementioned general lack of compositionality of the analyzes for different applications

7.5. Extensions

The approach described in the previous sections represents the core idea behind the solution. Different improvements and specializations are possible. This section describes the most relevant ones, and the price they would come at.

Firstly, as shown in Listing 7.2, the report generated by the static analysis phase is currently *context-insensitive*. This means that it distinguishes sources and sinks by the name of their containing method and the bytecode offset of the corresponding instruction, but does not take into account any information about the *context* in which the containing method is invoked, and calling contexts may help in gaining precision. There exist several flavors of context-sensitivity, and for JOANA, the kind of contexts which can be incorporated into the report is determined by the kind of points-to analysis (see Section 7.2.1) used. For simplicity's sake, the following example uses 1-CFA to illustrate the potential for precision gains of context-sensitive analysis reports. Consider the following code snippet:

```
1  class A {
2   public m(int x){
3    output(x);
4   }
5  }
```

```
6  class B {
7   public main(int i){
8    A a = new A();
9    x = input();
10   if (i == 0)
11       a.m(x);
12   a.m(42);
13  }
14 }
```

Let assume that at runtime the method `main` is only executed with parameter $i \neq 0$. Hence, method `A.m` is only invoked from line 12 which means that sink t in line 3 is not influenced by source s in line 9. JOANA is able to determine this statically, if run with 1-CFA as points-to analysis. In this example, it would distinguish two contexts of t: The context c_1 in which `A.m` is invoked from line 11 and the context c_2 in which `A.m` is invoked from line 12. Due to context-sensitivity of the used slicing algorithm, JOANA then deduces that there may be a flow from s to t in c_1 but no flow from s to t in c_2.

In a context-insensitive flow report, in contrast, all contexts of t would be merged: There would be only one instance of t and a flow from s to t would be reported simply because there exists at least one context c of t (namely c_1) such that t may be influenced by s if it is executed in c.

Of course, distinguishing between multiple instances of (i.e. contexts for) sources and sinks may result in a rise of the absolute number of reported flows. Nevertheless, there is a potential precision gain if at runtime the additional context information is also used to decide that a given sink is *not* influenced by a given source. It is also worth mentioning that, in order to make the instrumented code recognize the execution context of a cer-

tain sink at runtime, additional instrumentation is required. Depending on the kind of context-sensitivity, this may impact considerably on the runtime overhead. In order to keep the runtime overhead to a minimum, the proof-of-concept implementation of the tool described in this work compromises the possible additional precision in favor of the performance benefit of not requiring additional instrumentation.

Another way to make the analysis more precise is to "help" it by introducing manual declassification annotations or hand-refined source and sink specification. Although possible, this additional improvement has not been investigated in detail because the envisioned application context of this solution is a scenario where the static analysis is performed *automatically* and the code under analysis is possibly *unknown*.

The last possible extension that has not been discussed is the possibility of enforcing usage control requirements at the level of Java Bytecode in a *preventive* fashion [58], i.e. to execute a certain source/sink only if the tracker's response is affirmative. While implementation-wise this required only a minor addition to the instrumentation step, this line of work has been abandoned for two reasons: the instability of the system when blocking methods, and the difficulty to express higher-level usage control policies. Any interesting policy at the bytecode level that was identified during this research could be expressed and enforced in an easier way at different layers.

7.6. Challenges and Conclusions

This chapter described an approach to improve the precision of system-wide data flow tracking by integrating static information flow analysis results with runtime technologies. The proposed solution can track flows of data through an application and in-between different applications with a runtime overhead that, in the case studies, was better than similar approaches from the literature. At present, any claim of generalization of these results to other scenarios cannot be substantiated, but no fundamental argument against it has been found either.

While the proof-of-concept implementation discussed in the evaluation section connects executed Java code to an OS-layer runtime monitor, the general methodology presented in this chapter is not restricted to specific programming languages or tools, so instantiations for languages other than Java are possible. For instance, static approximations for flows in a data base could be connected to dynamic measurements in a Java application.

The work presented here is the first system-wide runtime analysis that replaces the internal behavior of applications by their static source/sink dependencies. Although hybrid approaches have already been proposed in the literature, this kind of integration of static and dynamic results is the first of its kind.

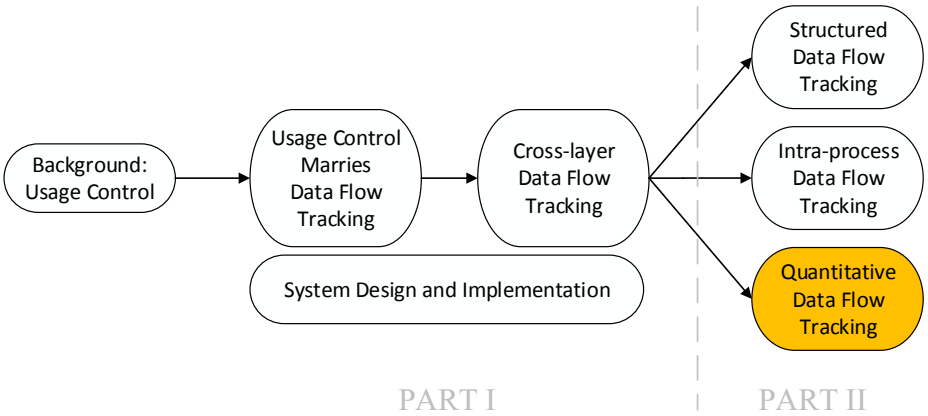
Experiments confirmed the intuition that the improvement of precision and performance depends on the type of information flows to consider as well as on the amount of I/O instructions compared to the total number of instructions. This solution is particularly suitable for situations in which this ratio is rather low, i.e. for those applications that perform large computations on small number of inputs and produce a limited number of outputs.

These results, detailed in the respective sections throughout the chapter, constitute valid and well-substantiated answers to the fundamental research questions presented at the beginning of this chapter.

Future research will focus on application of this work to other programming languages and the x86-binary level, although static analysis tools at this layer exhibit bigger limitations. Additionally, further investigations may provide a better understanding of the issues described in Section 7.5, in particular of context-sensitive analysis reports, which offer higher precision at the price of additional code instrumentation.

8. Quantitative Data Flow Tracking

This chapter presents a third extension of Chapter 3's model. In this work, co-authored by the author of this dissertation and published in [110], the basic model is augmented with information about the amount of data transferred by system events. This supports more expressive policies and containers declassification criteria to mitigate overapproximation issues.



8.1. Introduction

The first part of this work already discussed how data flow tracking concepts can be exploited to support usage control. Using the model presented in Chapter 3, 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.

This chapter proposes a refinement of this model that keeps track of the (estimated) *amount* of data that flows to a container: it does not limit the analysis to whether or not data flowed, but it tries to answer the question of *how much data flowed?*

The fundamental research questions addressed in this chapter are

How can the precision of the data flow tracking model be augmented with additional information about the amount of data?

How can the amount of data in the different representations be quantified and how can it be used to improve precision?

How can the improvement in terms of precision be quantified?

What is the overhead induced by the solution?

Which aspects of the system does it depend on?

8.1.1. Motivation

In information flow analysis, quantitative measurements are usually applied to source codes, in order to determine how many bits of information are transferred from some specific inputs to some specific outputs by one, some or every possible execution of a program. Similarly to the extension described in Chapter 6, these approaches cope with both explicit and implicit flows, due to the presence of a control-flow. But when this information is not available, only explicit flows can be measured, and that is the general case of the system considered in this chapter.

One may object that quantitative information-flow analysis without implicit flows is not significantly challenging; that is only partially true. Consider, for instance, a representation resulting from the merging of other two; as explained in Section 8.3, a precise estimation of the amount of data stored in the merged representation may require the whole history of every transfer of the secret across the system. And even though every kind of information can be measured in bits, different encodings may result in significantly different sizes for representations of the same data, especially when representations reside at different layers of abstractions. Note that the work described in this chapter focuses on quantitative measurement techniques that assume just one type of data (fixed encoding, no compression); the issues related to the more general case of different encodings are discussed in Section 8.5.4.

The benefit offered by a correct measure of the amount of data stored and flowing across different representations is twofold: on one hand it allows specification of *quantitative policies* such as "*if a file contains more than 10% of data from the customer database, then it must be deleted on logout*"; on the other hand, it lays the basis for reasonable declassification criteria, like in the exemplary scenario described at the end of this section.

The other solutions proposed in the second part of this dissertation (see Chapter 6 and Chapter 7) aim at tackling the label creep problem (see Section 5.2.1) by refining Chapter 3's analysis in terms of false positives. Given the same initial state of data dissemination and the same trace of events, these models leverage additional information about the system in order to reduce the number of containers that *possibly* contain the data after the execution of the trace; in contrast, the approach described in this chapter would result in the same number estimated by the basic model. The improvement in terms of label-creep mitigation stems from the possibility of applying the restriction associated to the sensitive data *only to those containers that contain a certain amount of such data*, e.g. "*if a file contains more than 10% of data from the customer database, then it must be deleted on logout*". In this sense, a file containing less than 10% of customer data is equivalent to a file that contains no customer data, reducing the impact of the restriction over system functionality when the taint propagation analysis is too conservative.

Quantitative measurements are useful for preventive enforcement but also for *a posteriori* analyses (see Section 1.3). 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, sometimes this may be the only way to get the job done in time, in spite of the policy violation. 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 the model for quantitative data flow tracking presented in this chapter (*quantitative model*, from now on) it is possible to measure quantitative flows of data and to enforce rules on them.

The quantitative model provides information about the amount of data stored in representations for usage control purposes. Nevertheless, it could be used also in other contexts or in combination with other techniques (e.g. risk analyses).

Problem. This work investigates the problem of measuring flows of data and how to embed such measurements into the usage control framework described in the first part of this dissertation.

Solution. This chapter defines a formal model for dynamic quantitative information flow measurements that (i) extends the basic tainting approach with quantitative measurements and (ii) augments the expressiveness of the existing model by allowing specifications of *quantity-based* policies.

Contribution. The contribution of this work is a generic layer-agnostic model for quantitative data flow tracking that can be instantiated to different layers of abstraction and that uses a non-probabilistic layer-specific quantitative measure for data (*units*). An exemplary instantiation of the model at the OS-layer is also implemented and evaluated.

8.1.2. Example Scenario

> *Alice works as an analyst for a smartphone manufacturer. Her job includes combining information about new models under development 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-layer 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.*

8.2. Measuring Data Quantities

The goal of a data flow tracking model is to track the distribution of *data* across different *containers*. If some data is stored in a 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. Following this intuition, the model for quantitative data flow tracking presented here (from now on, *quantitative model*) refines the so-called *tainting approach* (yes, data is stored in this container / no, it is not) by the notion of *quantity* (5Mb of data are stored in this representation). Given a specific container, such as a file or a database, knowing *how much* different sensitive data is stored in it helps enforcing policies such as *if more than 1MB of data related to a phone specification is stored in a file, then that file must be encrypted and deleted on log out*.

Let *data unit* be the smallest part of a container that can be addressed by an event of the system, like a system call or a query, and let *size* of a container be 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 layer of instantiation. For example, at the database layer, records (or cells) are units ("*database d.db contains 14 sensitive records*") whereas at the file system layer, 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 flows across different layers, units at one layer must be converted into units of the other layer. Conversion issues are discussed in Section 8.5.4.

The goal of this quantitative model 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 the 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), the quantitative model estimates how many *different* units of a certain data item are stored in each container of the system at a specific moment in time. This means that 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.

Each unit is assumed to be as informative as any other unit, i.e. no unit is more important than another. If this assumption cannot be justified, then the data items must be structured into parts of different informative value, and that each part is then tracked separately, possibly leveraging information about special events in the system, as discussed in Chapter 6. Covert or side-channels are not considered (e.g., knowing whether a unit has been copied or not does not leak any information about its actual value). Compression

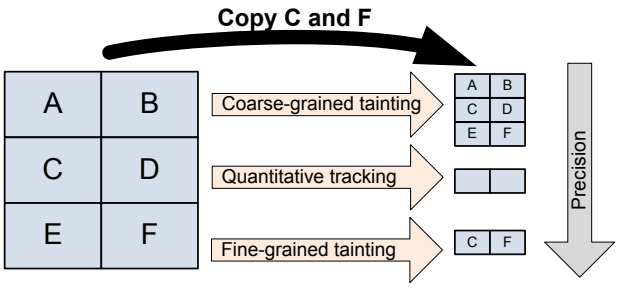


Figure 8.1.: Copying two blocks C and F: coarse-grained tainting (equivalent to copying the whole container); quantitative tracking (two blocks are copied without knowing which ones); fine-grained tainting (knowing copied blocks) [110].

and/or different encoding schemas conflict with the assumption of informative equality of units. However, in closed or semi-closed environments as in Section 8.1.2's example, it is reasonable to assume the behavior of every single event or process in the system to be known, and therefore also where compression takes place. For simplicity's sake, this case is not considered in the presented solution (see Section 8.5.4 for more details).

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).

Figure 8.1 illustrates how the precision of the quantitative approach discussed in this chapter lies in-between the precision of 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 it. Fine-grained tainting tracks each data unit independently and computes exactly which data unit is stored in each container. The quantitative method proposed here 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.

8.3. Quantitative Data Flow Tracking

This section describes a model to compute for every container, at each moment in time, an upper bound for the number of different units of a sensitive data item stored in it.

The model relies on three abstractions: data, containers, and events. For simplicity's sake, this approach considers only three types of events affecting 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 instanti-

ated according to the instantiation of the model (e.g., system calls for an OS, queries for a database). After introducing this simple model, its relation to the one presented in Chapter 3 is discussed.

Initially, each container in the system contains no unit of sensitive data. If an event introduces new sensitive data into the system, the event is modeled by mapping the number of new sensitive units of data to its initial representation(s). Usually, the initial amount of sensitive data corresponds to the size of the initial representation, but the model allows 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 data d, every event in the system that corresponds to a transfer or deletion of units of d in some container is monitored. The remaining of this section shows how different events may 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, it is important to keep track of previous transfers. In the model, this historic information is stored in *provenance graphs*.

8.3.1. Provenance Graphs

For simplicity's sake, the remaining of this section assumes that there exists only one data item d in the system. Multiple data items can be handled with several provenance graphs, one per item, as discussed in Section 8.4.

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: $\mathcal{N} \subseteq \mathcal{C} \times \mathbb{N}$, where $\mathcal{C}$ is the set of containers. $\mathcal{N}$ also contains 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.

The **goal** of the provenance graph is to support the computation of an upper bound for the number of different sensitive units of data d in each container in the system. For reasons that will become apparent in step 6 of the example below, this is computed on the grounds of a provenance graph. Section 8.3.2 defines κ, the function that computes such

an upper bound for those nodes of the provenance graph that correspond to the containers in the current (that is, latest) timestep.

8.3.2. Runtime Construction

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 is discussed below). For every event at time t, at most one new node and 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 $\mathcal{G}_t$.

The following are possible graph evolutions:

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, events of this type add a node (c_i, t) to $\mathcal{N}_{t-1}$ and, for d being composed by m units of sensitive data, 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 the model adds 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 data is copied from the "latest representation" of container c_1).
- **C2 step** If c_2 already exists (copying then is appending), its content at time $t - 1$ also needs to be considered. In this case, there exists 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, the edge $\big((c_2, t'), \kappa((c_2, t')), (c_2, t)\big)$ is added to $\mathcal{E}_{t-1}$. Replacing the label $\kappa((c_2, t'))$ by ∞ would not impact correctness nor precision; the choice of $\kappa((c_2, t'))$ is motivated by presentation concerns. Remember that $\kappa((c_2, t'))$ is an upper bound for the number of sensitive units in c_2 at time t' and therefore also an upper bound for the number of sensitive units that can flow from (c_2, t') to (c_2, t).

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

At each moment of time, provenance graphs are defined to be the smallest relations satisfying the above three properties. 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.

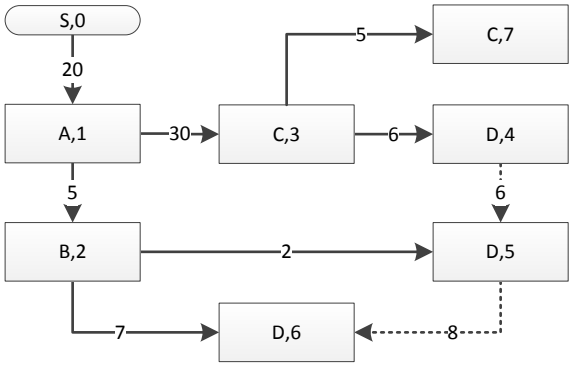


Figure 8.2.: Example provenance graph at time $t = 6$. Dashed arrows are C2 steps.

8.3.3. Step-by-step Example

Consider the example in Figure 8.2.

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 labeled 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.

2. In the second step, 5 units of data are copied from A to B, which results in node $(B, 2)$ and the edge labeled 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 labeled 6 from node $(C, 3)$ to $(D, 4)$. As discussed in the previous point, C contains at most 20 sensitive units, and a conservative estimation must assume that all data copied from C to D is sensitive, thus $\kappa((D, 4)) = 6$.

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 labeled 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 $(D, 5)$ labeled 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). This creates a new edge labeled 7 from $(B, 2)$ to a new node $(D, 6)$, and an edge labeled $\kappa((D, 5)) = 8$ 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, as discussed at the beginning of this section, 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 estimation 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 these units do not need to be counted 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 labeled 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.

8.3.4. Rationale

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 can be known 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 explained below). It is this historical knowledge that allows to reduce the upper bound $\kappa((D,6))$ from 13 to 11 units.

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, the upper bound would still be $\kappa((D,6)) = 11$. 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. They can hence be interpreted 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 the 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 $(D,4)$.

8.3.5. Computation of κ

The definition of κ then is simple: if $n \in \mathcal{N}_t$ is the new node created in timestep t, then $\kappa(n) = maxflow_{\mathcal{G}_t}((S,0),n)$. Since every container can be the destination of a copying event in the next step (a C step), the κ value of any node needs to be available at any moment in time.

Algorithmically, this can be done by computing at most only one max-flow/min-cut on a directed acyclic graph per event (for the new node): because the model never adds

incoming edges to existing nodes, their κ value is constant over time and does not need to be recomputed.

8.3.6. Correctness

The 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 computed κ value. Let φ be the oracle function that provides this actual amount for a node (i.e. container and moment in time). Note that φ is not defined but just assumed to exist.

Let also assume all events in the system to be adequately reflected in the construction of the flow graph. Whenever an event moves data in-between containers, such event is used to construct the provenance graph according to the description above. Conversely, no edge or node is added without the 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 $\forall t \in \mathbb{N} \forall c \in \mathcal{C} : \varphi(c,t) \leq \kappa((c,t))$ indeed holds is provided in Section C.1.

8.3.7. Simplification

Because the complexity of the computation of maximum flows depends on the size of the graph, it is desirable to keep graphs small. Obviously, any reduction in size should preserve the correctness and the precision of the algorithm presented above. The *simplification rules* described in the following 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 $\kappa'(n') = maxflow_{\mathcal{G}'_t}((S,0),n')$;

(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 the set of *current nodes* $cn_{\mathcal{G}}(X)$ represent the nodes in $\mathcal{G}$ that are the latest representation of each container in $X \subseteq \mathcal{C}$:

$$\forall \mathcal{G} = (\mathcal{N}, \mathcal{E}), X \subseteq \mathcal{C} : cn_{\mathcal{G}}(X) = \{(c,t) | c \in X \wedge (c,t) \in \mathcal{N} \wedge \forall t' : (c,t') \in \mathcal{N} \implies t' \leq t\}.$$

Assuming a set of containers X, $\mathcal{G}_{t,X}$ denotes the graph in which every node in $cn_{\mathcal{G}_t}(X)$ is connected to a dummy node dn that represents a virtual sink of all future operations on the nodes in $cn_{\mathcal{G}_t}(X)$ (and therefore all future evolutions of $\mathcal{G}_t$ on every possible set of containers):

$$\mathcal{G}_{t,X} = (\mathcal{N}_t \cup \{dn\}, \mathcal{E}_t \cup \bigcup_{n \in cn_{\mathcal{G}_t}(X)} \{(n, \infty, dn)\}).$$

Definition 8.1 (Equivalence of Provenance Graphs). *Let* $\kappa_{t,X}(n) = maxflow_{\mathcal{G}_{t,X}}((S,0),n)$ *for any node* n*, and, similarly,* $\kappa'_{t,X}(n) = maxflow_{\mathcal{G}'_{t,X}}((S,0),n)$*. Equivalence of provenance graphs is then defined by the following invariant:*

$$\mathcal{G}_t \sim \mathcal{G}'_t \iff \forall X \subseteq \mathcal{C} : \kappa_{t,X}(dn) = \kappa'_{t,X}(dn)$$

The following are the most relevant cases identified during this research where, given a provenance graph, a smaller but equivalent provenance graph could be obtained. Small graphs considerably increase performance in the experiments of Section 8.5.2. 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 C.

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. Thus, the simplification consists in modifying the edge label of the copying step at time k, replacing any further copying action from c_b by a copying action from c_a, and removing 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. Thus, the simplification consists in adding c_c to the source containers of the copying at time k and removing 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. The simplification in this case consists in replacing the amount transferred at time k (s_1) with the value $s_1 + s_2$ and removing the last copying step.

8.4. Quantitative Policies

8.4.1. Semantic Model

Provenance graphs are now used to generalize the model presented in Chapter 3 by combining quantitative data flow tracking with data usage control.

In the quantitative model, the state of the system is given by the association between each data item and its provenance graph ($\Sigma_q = \mathcal{D} \to Graph$). Provenance graphs consist of nodes, $Nodes \subseteq \mathcal{C} \times \mathbb{N}$, i.e. container-timestamp pairs. $Nodes$ contains a reserved identifier $(S, 0)$ that stands for the external source of data for each graph. Provenance graphs are modeled as a (partial) function of type $Graph = Nodes \times Nodes \to \mathbb{N}$ 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 $step : (Graph \times \mathcal{S}) \to Graph$ updates the graph according to the rules presented in Section 8.3. The quantitative model assumes 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. In real implementations, however, each system event may correspond to a sequence of possible graph evolutions, e.g. "move A to B" corresponds to a transfer form A to B followed by the truncation of A to 0. For simplicity's these cases are assumed to be handled internally by the $step$ function, mapping each system event to the respective sequence of graph evolutions, applying all of them to the input graph and returning only the final result.

Then, $\mathcal{R}_q : (\Sigma_q \times \mathcal{S}) \to \Sigma_q$ models the application of event e to each provenance graph in σ. This is done by function $step$: if $\mathcal{R}_q(\sigma, e) = \sigma'$, then $\sigma'(d) = step(\sigma(d), e)$ for each data item $d \in \mathrm{dom}(\sigma)$.

Quantitative usage control policies are defined over traces that map abstract time points to events ($Trace : \mathbb{N} \to \mathbb{P}(\mathcal{S})$).

Given a trace t, function $states_q : (Trace \times \mathbb{N}) \to \Sigma$ computes the information state at a given moment in time n as

$$states_q(t, n) = \begin{cases} \sigma_i & \text{if } n = 0 \\ \mathcal{R}_q(states_q(t, n-1), t(n-1)) & \text{otherwise} \end{cases}.$$

Note that, as for the basic model, the assumption of independence of events within the same timestep is assumed to hold (see Section 2.1.3). For this reason, in the definition of $states_q$, $\mathcal{R}_q$ has been overloaded to handle sets of events (in form of any arbitrary serialization).

8.4.2. Policies

A general discussion on usage control policies can be found in Chapter 3. Here, the focus is on one possible way of restricting data quantities. To this end, the model needs to capture function κ introduced in Section 8.3. Given a graph $\mathcal{G}_t$ and a container c, function $K : (Graph \times \mathcal{C}) \to \mathbb{N}$ 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 $\mathcal{G}_t$.

Let Φ_q be the set of operators to specify quantitative policies, defined as

$$\Phi_q ::= atMostInEach(\mathcal{D}, \mathbb{N}, \mathbb{P}(\mathcal{C})) \mid atMostInSet(\mathcal{D}, \mathbb{N}, \mathbb{P}(\mathcal{C}))$$

where

- `atMostInEach(d, q, C)` specifies that at most q units of data d may flow to any of the containers belonging to set C. For instance, the policy *"No outgoing mail can contain more than 10KB of sensitive data"* is, at each moment in time, expressed as `atMostInEach(d, 10KB, MAILS)`, where MAILS is the set of containers that represents outgoing mails.
- `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 `atMostInSet(d,1MB,REMOVABLE)`, where `REMOVABLE` is the set of containers that represent files on removable devices.

Given a trace and a time point, the semantics $\models_q \subseteq (Trace \times \mathbb{N}) \times \Phi_q$ of these quantitative data usage operators is defined by

$$\begin{aligned}
&\forall t \in Trace, \forall n \in \mathbb{N}, \forall \phi \in \Phi_q, \forall \sigma \in \Sigma : (t,n) \models_q \phi \iff \\
&\sigma = states_q(t,n) \wedge \exists d \in \mathcal{D}, Cs \subseteq \mathcal{C}, Q \in \mathbb{N} : \\
&\quad \phi = atMostInEach(d,Q,Cs) \wedge \forall c \in Cs : K(\sigma(d),c) \leq Q \vee \\
&\quad \phi = atMostInSet(d,Q,Cs) \wedge \textstyle\sum_{c \in Cs} K(\sigma(d),c) \leq Q
\end{aligned}$$

Using Φ_q only simple information flow policies can be specified. To express more complex policies, like those presented in Section 8.1, Φ_q must be embedded into the full temporal logic language defined in Chapter 3, obtaining the new language Φ_{iq}

$$\begin{aligned}
\Phi_{iq} ::= \; & (\Phi_{iq}) | \Psi | \underline{false} | \Phi_{iq} \; \underline{implies} \; \Phi_{iq} | \underline{forall} \; VName \; \underline{in} \; VVal : \Phi | \\
& \Phi_{iq} \; \underline{until} \; \Phi_{iq} | \Phi_{iq} \; \underline{after} \; \mathbb{N} | \underline{replim}(\mathbb{N},\mathbb{N},\mathbb{N},\Psi) | \underline{repuntil}(\mathbb{N},\Psi,\Phi_{iq}) | \\
& \Phi_s | \Phi_q
\end{aligned}$$

and its semantics $\models_{iq} \subseteq (Trace \times \mathbb{N}) \times \Phi_{iq}$

$$\begin{aligned}
&\forall t \in Trace; n \in \mathbb{N}; \varphi \in \Phi_{iq} : (t,n) \models_{iq} \varphi \Leftrightarrow \varphi \neq \underline{false} \wedge \\
&\big(\exists e \in \mathcal{VE} : (\varphi = E(e) \vee \varphi = I(e)) \wedge \exists e' \in t(n) : e' \models_e \varphi \\
&\vee \exists \psi, \chi \in \Phi_{iq} : \varphi = \psi \; \underline{implies} \; \chi \wedge \neg((t,n) \models_f \psi) \vee (t,n) \models_{iq} \chi \\
&\vee \exists \gamma \in \Gamma : \varphi = eval(\gamma) \wedge [\![\varphi]\!]_{eval} = true \\
&\vee \exists vn \in VName; vs \in VVal; \psi \in \Phi_{iq} \; : \\
&\quad \varphi = (\underline{forall} \; vn \; \underline{in} \; vs : \psi) \wedge \forall vv \in vs : (t,n) \models_{iq} \psi[vn \mapsto vv] \\
&\vee \exists \psi, \chi \in \Phi_{iq} : \varphi = \psi \; \underline{until} \; \chi \wedge \big(\forall v \in \mathbb{N} : n \leq v \Rightarrow (t,v) \models_{iq} \psi \\
&\quad \vee \exists u \in \mathbb{N} : n < u \wedge (t,u) \models_{iq} \chi \wedge \forall v \in \mathbb{N} : n \leq v < u \Rightarrow (t,v) \models_{iq} \psi\big) \\
&\vee \exists m \in \mathbb{N}; \psi \in \Phi_{iq} : \varphi = \psi \; \underline{after} \; m \wedge (t, n+m) \models_{iq} \psi \\
&\vee \exists m \in \mathbb{N}_1; l,r \in \mathbb{N}; \psi \in \Psi : \varphi = \underline{replim}(m,l,r,\psi) \wedge \\
&\quad l \leq \#\{j \in \mathbb{N}_1 | j \leq m \wedge (t, n+j) \models_{iq} \psi\} \leq r \\
&\vee \exists m \in \mathbb{N}; \psi \in \Psi, \chi \in \Phi_{iq} : \varphi = \underline{repuntil}(m,\psi,\chi) \\
&\quad \wedge \big((\exists u \in \mathbb{N}_1 : (t,n+u) \models_{iq} \chi \wedge (\forall v \in \mathbb{N}_1 : v < u \Rightarrow \neg((t,n+v) \models_{iq} \chi)) \\
&\qquad \wedge (\#\{j \in \mathbb{N}_1 | j \leq u \wedge t(n+j) \models_{iq} \psi\}) \leq m) \\
&\quad \vee (\#\{j \in \mathbb{N}_1 | t(n+j) \models_{iq} \psi\}) \leq m\big)\big) \\
&\vee \varphi \in \Phi_s \wedge (t,n) \models_s \varphi \\
&\vee \varphi \in \Phi_q \wedge (t,n) \models_q \varphi
\end{aligned}$$

Notice that the definition of Φ_{iq} and of its semantics are almost identical to those of Φ, except for the for the last line, which includes the state based operators from Φ_s. Using the Φ_{iq} language, it is then possible to specify policies such as $always(\neg atMostInSet(d, q, Cs) \Rightarrow notify)$ that issue a notification whenever the amount of data d stored in the specified set of containers Cs exceeds q. This kind of policies is used in the experimental evaluation in Section 8.5.

8.5. Evaluation

The adequacy of the quantitative model in terms of the scenario in Section 8.1 has been evaluated by conducting different sets of experiments, which measured precision and performance of the model alone, i.e. at an abstract level, and in concrete instantiations at the operating system layer (see Section 3.3.2), where the abstract events are associated system calls and API calls.

8.5.1. Implementation and Methodology

Remember that Alice performs a sequence of report generation/ update actions in the phone scenario presented in Section 8.1 and that she is subjected to a set of policies of the kind $always(\neg atMostInSet(d, 1MB, MAIL) \Rightarrow notify)$ (see Section 8.4). An action is a transfer of some units of data from a specification or from an existing report to another (possibly new) report. This experiment consists in generating random sequences of actions of different lengths, and observing the evolution of the model during their execution. In each step, Alice can choose between creating a new and updating an existing report with probability *PN*. The analyses in the remainder of this section refer to average and median values of the execution of 100 sequences for each length.

The first set of experiments (*atomic*, denoted by **A**), modeled each action after the initialization as one single abstract event, either a T or a C step (see Section 8.3), where specifications and reports are containers. The second set of experiments (*syscalls*, denoted by **S**), instantiated the model to the operating system layer (Linux) and performed quantitative data flow tracking at this layer. 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 $\mathcal{R}_q$ relation in Section 8.4. They also refine, by disjunction, the abstract events specified in the above policy.

The length of an event sequence depends on the size of the files and of the transfers involved. In the tests, one abstract action corresponds, on average, to 40 system calls. Each experiment started from the same arbitrary yet fixed amount of (sensitive) specifications and (initially non-sensitive) reports. For both experiments, the model is evaluated with

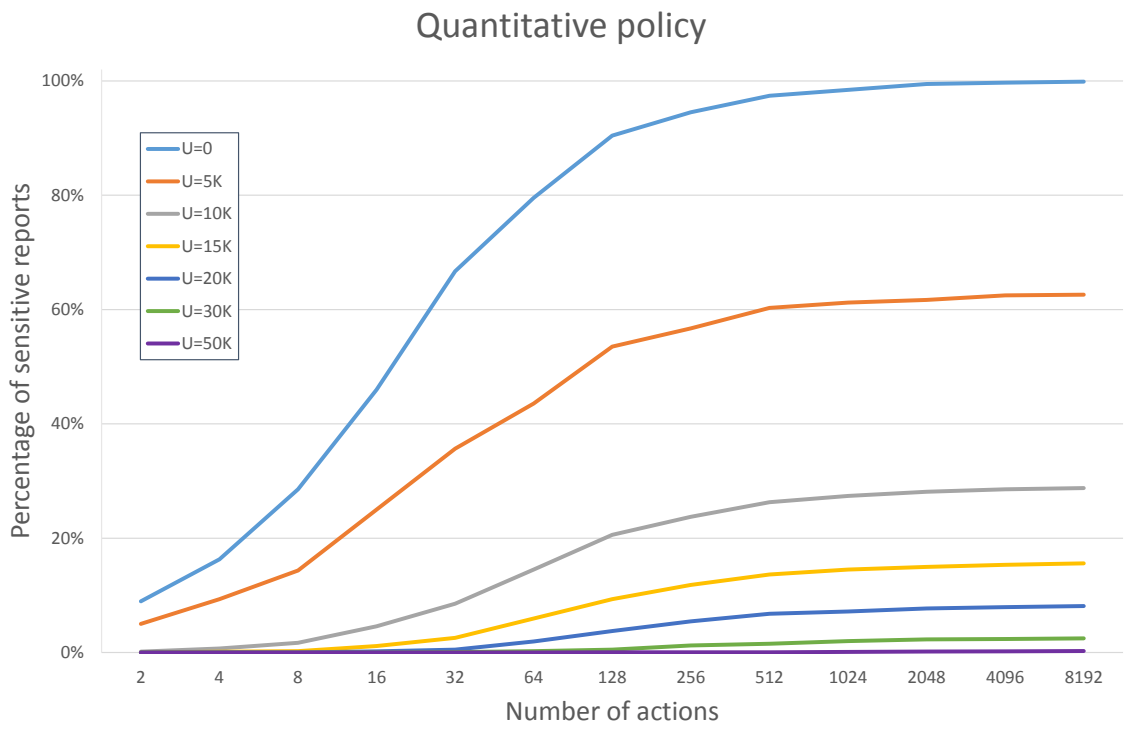


Figure 8.3.: Percentage of reports considered sensitive by a quantitative policy vs. number of actions (PN=50%)

(denoted **s**) and without (denoted **n**) simplification. The *syscalls* tests without simplification (i.e. the Sn configuration), consider only sequences of at most 1024 actions because the time required to perform 100 executions longer than 1024 is prohibitive without simplification.

In the second set of experiments, the quantitative model is instantiated at the operating system layer. The amount of bytes that a process tries to read/write from/to a file corresponds to the event size used by the 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.

The experiments address the following questions:

RQ1 How precise is the quantitative model (i.e. how far is the estimated value from the exact amount of different sensitive data units)?

RQ2 What is the overhead induced by the quantitative model w.r.t native execution?

8.5.2. RQ1 - 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", see Section 5.2.1). For a typical run of the system, Figure 8.3 shows the relative number of reports considered sensitive (tainted) for the policy

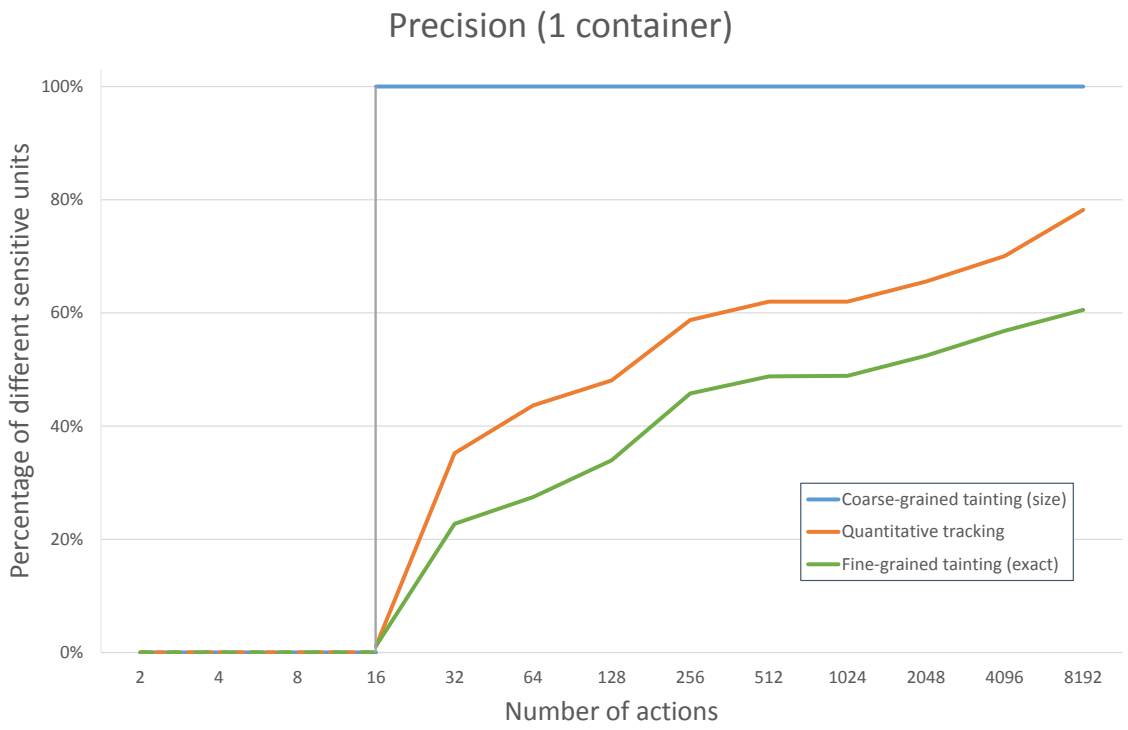


Figure 8.4.: Estimation of different sensitive units in an exemplary container during one execution (PN=50%). Before the 16th action, no sensitive data is stored in the container.

"*do not distribute a report if it contains more than U units of sensitive data*", for different values of 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).

A second perspective is provided by Figure 8.4, which shows how many units are considered sensitive according to different tracking methods for a typical container. As expected, the quantitative estimation is in-between coarse-grained and fine-grained tainting.

To more precisely answer the first research question it is necessary 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 the model is. Therefore, to empirically measure the precision of the model for a container c at a specific moment in time, let precision be defined by function $Prec$ as

$$Prec(c) = \begin{cases} 1 & \text{if } Size(c) = Exact(c) \\ \frac{Size(c) - Estim(c)}{Size(c) - Exact(c)} & \text{Otherwise} \end{cases},$$

where $Size(c)$ is the total number of data units in c (including non-sensitive data and duplicates), $Exact(c)$ is the precise number of different sensitive units in c, and $Estim(c)$ is the number computed by the model, i.e. the value of κ. Thus, $Exact(c) \leq Estim(c) \leq Size(c)$. The estimation is most accurate if $Estim(c) = Exact(c)$ (i.e. $Prec(c) = 1$), and the

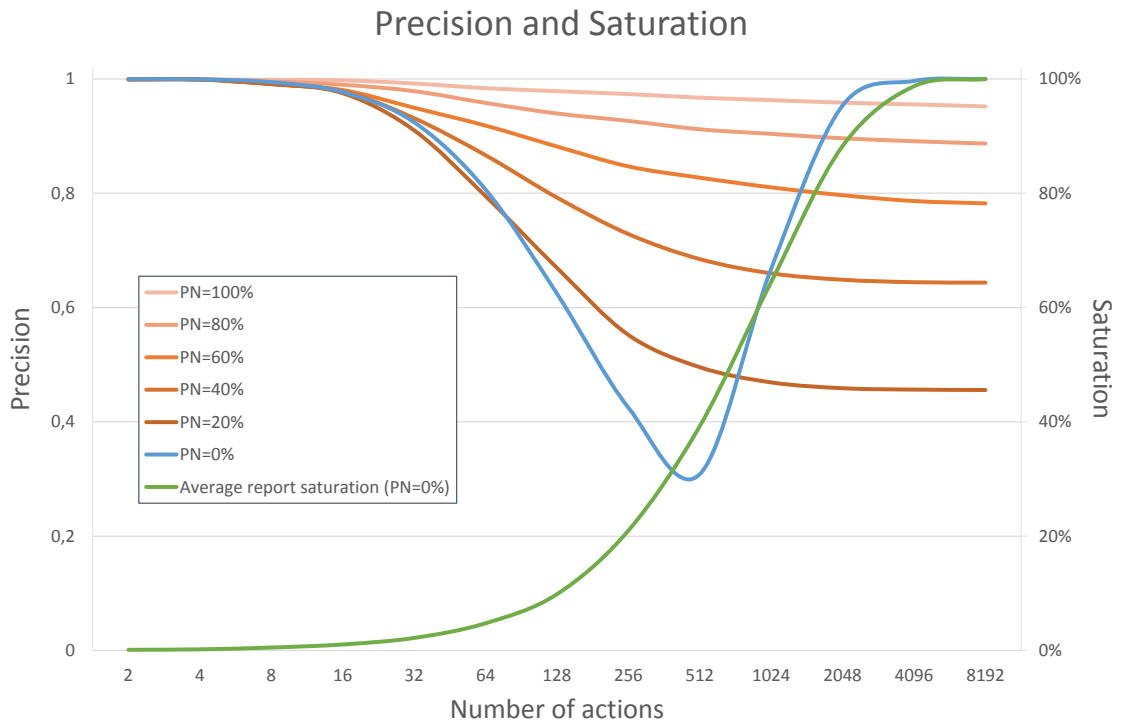


Figure 8.5.: Median precision vs. number of events and percentage of new reports for *syscalls* tests

worst case ($Prec(c) = 0$) is $Estim(c) = Size(c)$. In the special case $Size(c) = Exact(c)$, the estimation is still correct (i.e. $Prec(c) = 1$). The proportion of non-sensitive units (or duplicates of sensitive units) in a container c that are incorrectly considered additional different sensitive units by the model corresponds to $1 - Prec(c)$.

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 8.5 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, in this test the probability PN that she creates a new report in each step varies 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 ($Exact(c) = max$), such report is *saturated*, because no more

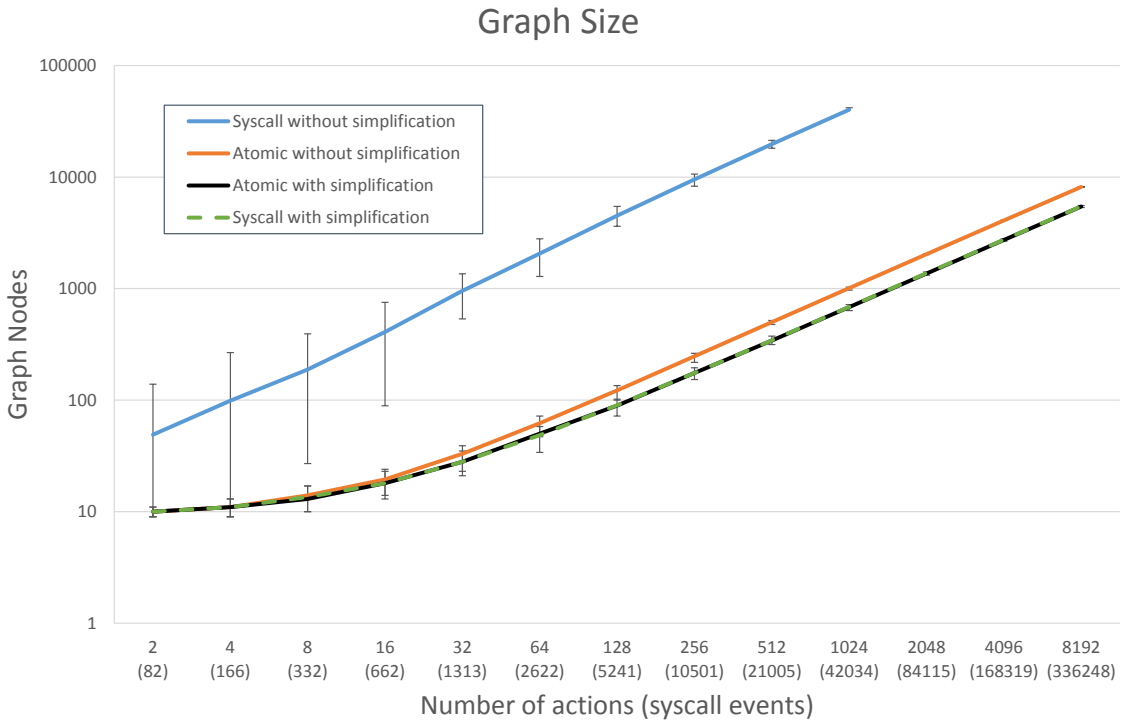


Figure 8.6.: Graph size vs number of actions (PN=50%), layer of abstraction (**A**tomic/**S**yscall), with (**s**) and without (**n**) simplification. Note that As and Ss overlap.

different sensitive units can be added to it. Considering that max is an upper bound for the estimation, precision for saturated containers is equal to 1 ($max = Exact(c) \leq Estim(c) \leq max \implies Exact(c) = Estim(c)$), and the closer to saturation a container is, the higher its precision will be. Figure 8.5 shows the positive correlation, after some time, between the median precision (dotted line) and the average report saturation (average $(Exact(c)/max)$ 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 the precision.

Simplification

Without simplification (Section 8.3), 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 operating system layer requires observing many events (~ 40 system calls per action), which significantly impacts the size of the provenance graph.

Figure 8.6 shows the growth of the provenance graph as the number of actions increases. This test compares the monitoring of each action as an atomic transfer (1 action = 1 event) with the instantiation at the OS-layer (1 action ~ 40 events), with and without simplification. By construction, the graphs grow by at most one node after each event. As expected,

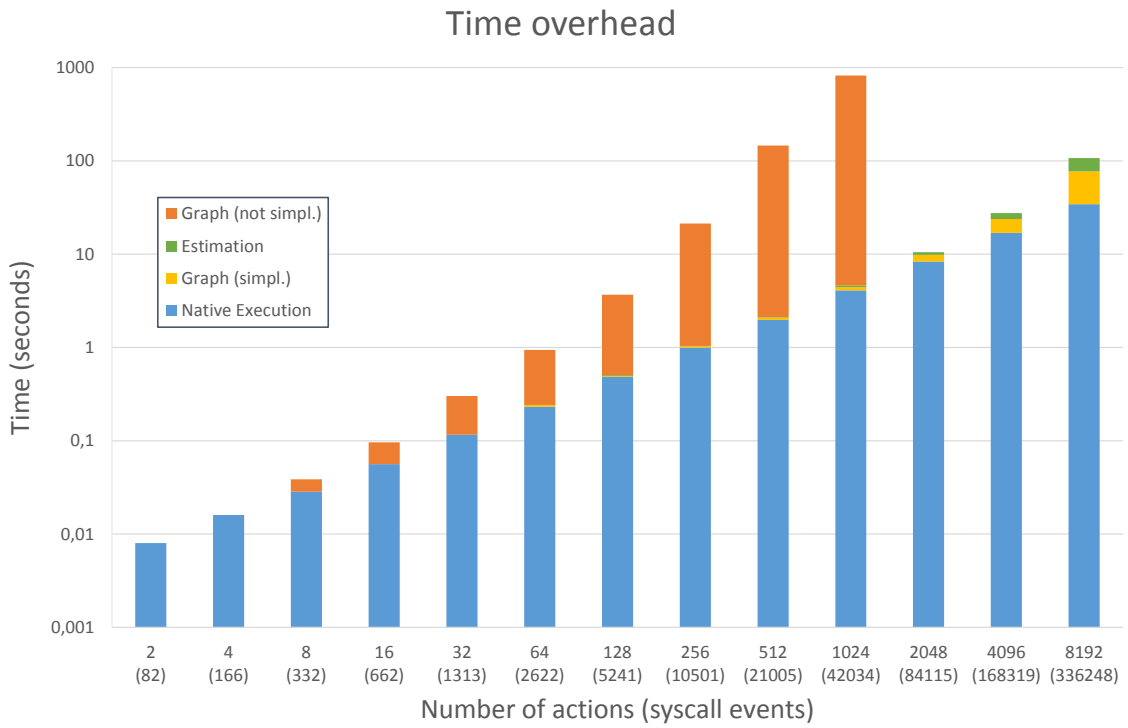


Figure 8.7.: Time overhead for the *syscalls* tests (PN=50%) with and without QDFT, with and without optimizations.

monitoring a forty-fold number of events is reflected in a graph forty 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.

8.5.3. RQ2 - Performance

In addition to precision, it is also interesting to evaluate the storage and the time required to update the quantitative model when receiving an event. As for the graph's size, performance depends on the sequence of actions performed by the user. The implementation requires a fixed amount of ∼12Mb of RAM plus ∼285 bytes per node and ∼110 bytes per edge, i.e. less than 17Mb in total for traces of more than 330K events. Because these numbers are negligible on modern machines, the performance analysis focuses on the time dimension.

Worst-case scenario

As mentioned in Section 8.3.2, it is possible for every **C2** step to replace the edge's capacity by ∞ without impacting correctness or precision. This allows 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 the considered scenario this means computing max-flow only upon observing 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, the performance overhead of maintaining the provenance graph with and without applying optimizations (simplification rules and lazy evaluation) is measured. Figure 8.7 shows the median performance overhead compared to the native execution time. For the optimized case, Figure 8.7 distinguishes 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*, another experiment has been conducted: the quantitative model has been instantiated for another operating system (MS Windows), where events are API calls (`WriteFile()`, `ReadFile()`, `CreateFile()`, etc.) and containers are files, pipes and memory locations. As the previous experiments, this one considered a set of sensitive source files and the creation of 64 reports with PN=50%. This scenario simulates 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.

With respect to the previous experiments, this scenario shows two important differences: a) in addition to the API calls generated by the user interaction, every other API call in the system is monitored, 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* and the *atomic* experiments described before.

For this experiment, a total of $60K$ API calls has been observed, out of which $48K$ were filtered using trivial heuristics (like ignoring certain system services). The remaining 12K were circa 10% related to sensitive data and 90% unrelated "noise" calls created by other processes. All the 12K calls were sent to the quantitative 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 the model to handle all the updates was $\sim$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 perceivable. This further instantiation of the model shows

that it is indeed usable in a realistic scenario.

8.5.4. Discussion

The experimental results confirm that the precision of the quantitative model 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 quantitative estimations in the use cases considered 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 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) is observed. 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 the 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 the 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 the traces used for experiments represented 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% [73], and 1-3 orders of magnitude for other layers of abstraction [136]), the delay introduced by the quantitative model is absolute, i.e. *independent* of the time required to *perform* or *intercept* the actions. This means that, for instance, $1ms$ to update the provenance graph may be an intolerable overhead when modeling the execution of a system call, but may at the same time be a negligible delay for a BPEL or ESB event [126].

8.6. Challenges and Conclusions

This chapter presented an extension of the model for data flow tracking discussed in Chapter 3 that leverages information about quantities of data stored in containers. It proved the correctness of the model and assessed its precision and performance by discussing an implementation of it and embedding it in the usage control framework presented in the first part of this dissertation.

The precision of the result depends on the specific actions performed by the user, and can either be as good as that of a fine-grained approach that knows exactly how many different units of each data are stored in every container, or as as bad as that of a coarse-grained approach which is only aware of the possibility of data being stored in a certain container, but without any information about the amount of it. Usually, the precision of the quantitative model lies somewhere in between the two by over-approximating the exact amount of sensitive units.

The results described in this chapter, in particular the integration of data quantities in the model, the definition of tracking precision in Section 8.5.2, and, in general, the evalu-

ation performed in Section 8.5 satisfactorily address the fundamental research questions discussed at the beginning of this chapter.

Augmenting the data flow tracking model with the quantitative features described in this chapter allows the framework 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 the quantitative 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.

Some of the limitations of this work that will be addressed as future work are: coping with compression, modeling sources of data of variable sizes and investigating the problem of measuring data across different layers of abstraction (i.e. converting units of data), which at the moment can only be treated in very special cases (i.e. when representations are compressed/converted using fixed-ratios). If compression of multiple files into a single archive is a common event in the scenario, maybe a model that leverages structure of data (see Chapter 6)) rather than its amount could provide better results.

9. Related Work

This chapter relates the work presented in this dissertation to similar work from the literature. Many of the results described in this dissertation have been presented and published at international conferences and workshops; Section 9.1 presents an overview of them, whilst the following sections put the results in the context of existing solutions from the literature. Relevant literature is organized by separating related works in terms of usage control (see Section 9.2) from related work in terms of information flow tracking, in turn distinguishing between approaches based on static (see Section 9.3.1), dynamic (see Section 9.3.2) and hybrid (see Section 9.3.3) analyses.

9.1. Author's Prior Work

All the conference publications cited in this section are co-authored by the author of this dissertation.

The model described in Chapter 3 is the first model that combines event-driven usage control with data flow tracking for an arbitrary layer of abstraction. The architecture of the framework (see Chapter 5) has been published in [100], while the complete model for an arbitrary layer of abstraction has been firstly presented in [138]. At the time of writing, the cross layer model presented in Chapter 4 is submitted for peer-reviewing [109].

While all the three extensions presented in the second part have been developed more or less in parallel, the model for quantitative data flow tracking (see Chapter 8) was the first to be published [110], whereas a preliminary version of the model for structured data flow tracking (see Chapter 6) was published just few months later [108]. Chapter 7 discusses in detail the idea of replacing dynamic intra-process data flow tracking with a simple monitor for input and output instructions that leverages static information flow analysis result. This idea has been initially investigated in the context of Android applications [141], but the analysis described in this dissertation tackles the problem from a more general perspective that subsumes the results presented in [141]. At the time of writing, such more comprehensive work has been accepted for publication [107]. Lastly, the author also discussed how to secure inter-layer communication for usage control within a hypervisor in [120] and [121].

The implementations of the monitors for the MS Windows operating system has been developed within the context of a Master thesis's work [165] supervised by the author, as well as the monitor for MS Excel [149], and a number of monitors for other layers of abstraction [163, 105, 27].

9.2. Usage Control

The major contribution of the this thesis's work is the combination of data flow detection with usage control and the formalization of a general-purpose policy specification language for usage control.

In terms of general-purpose usage control models, the model presented in Chapter 3 exhibits some similarities with the models underlying XACML [143], Ponder2 [154] and UCON [129]. The first two, however, do not provide formalized support for cardinality or temporal operators (free text fields exist, but the respective requirements are hard to enforce). UCON supports complex conditions [174], and has been used in applications at different system layers, such as the Java Virtual Machine [124], the Enterprise Service Bus [62] and cloud federations [16], but assumes that data never leaves the data provider's realm, implicitly making UCON policies device-dependent.

Enforcement of usage control requirements has been done for the OS layer [48, 170, 73, 21, 130], for the X11 layer [136], for Java [39, 86, 50], the .NET CIL [45] and machine languages [52, 166, 41]; at the level of an enterprise service bus [63, 126]; for dedicated applications such as the Internet Explorer [49] and in the context of digital rights management [12, 116, 133]. These solutions focus on either data flow tracking or event-driven usage control. The model described in this dissertation, in contrast, tackles both dimensions of the problem at the same time and since it is layer-independent, it can be instantiated to each of these layers. An exception is the work of Kelbert et al. [91, 92], that adapted the generic model of Chapter 3 to distributed systems, in order to face the additional challenges this domain presents (see Section 5.1.4).

At the level of binary files, the Garm tool [41], described in more detail in Section 9.3.3, combines data tracking with an enforcement mechanism for basic usage control. Garm focuses on access control, trust and policy management aspects, while the goal of the work presented in this dissertation is a generic model and a policy language to express and enforce advanced usage control requirements for arbitrary system layers. Data flow confinement is also intensely studied at the operating system layer [48, 170, 130]; here, this work differs in that it aims at enforcing complex usage control policies.

Note that, in addition to purpose and environmental constraints (see Section 1.1), the language described in Chapter 2 assumes information about spatial context (e.g. GPS location) to be given as parameter of the events, and the respective semantics to be defined by an external operator (eval operator, see Chapter 2). Existing solutions from the literature [89, 20] offers detailed solutions to embed context information in the architecture and to implement a respective operator to evaluate spatial constraints; in contrast to this framework, however, these solutions are tailored to mobile devices contexts and do not aim at a generic solution for arbitrary layers of abstraction.

A multitude of policy languages [47, 11, 18, 128, 173, 76, 40, 154, 160] has been proposed in literature, but none of them addresses the data dimension like this work does; they allow for definitions of usage restrictions for specific rather than all representations of data, and their semantic models do not consider data flows. A remarkable exception is the work of Kumari et al. (built on top of the results presented in the first part of this dissertation [138]), who developed a framework for policy translation [98, 99] to transform policies described in high-level domain-specific language into implementation level policies as described in Section 2.3.

Complex event processing [112] and runtime monitoring [102] are suitable for monitoring conditions of usage control policies. As such, they address one aspect of the problem, namely the monitoring part, and do not cater to data flow. Concepts from the state of the art in these disciplines has been used in this work to support efficient runtime monitoring of usage control policies.

9.3. Information Flow Control

Information flow tracking, in the context of security policy enforcement [147], is at least thirty years old [53]. The two main approaches are *static* [42, 119, 85] and *runtime* checking, based on dynamic tainting analysis. This section discusses the position of this work with respect to static and dynamic approaches in Section 9.3.1 and Section 9.3.2, respectively. More recently, many solutions tried to combine these two approaches in order to compensate the limitations of one with the strengths of the other. Such *hybrid* approaches and their relation to this work are discussed in Section 9.3.3

9.3.1. Static Approaches

Static approaches analyze application code before it is executed and aim to detect all possible information flows [42, 157]. A given program is certified as secure, if no information flow can be found. Such a static certification can for example be used to reduce the need for runtime checks [43]. Various static approaches (apart from PDGs, already discussed in Section 7.2.1) can be found in the literature, which are usually based on type checking [157, 123, 131] or flow analysis [42, 15, 14, 22]. As discussed in Section 7.4, however, because of their nature, static approaches can hardly handle dynamic aspects of applications, like callbacks or reflective code, and are confined to the application under analysis. For this reason, this work focuses on dynamic and hybrid approaches.

Some relevant work, however, can be found in the field of static analysis when it comes to quantify information flows (see Chapter 8). In this sense, Denning [44] firstly 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., [34, 35]) rely on a specific entropy of the input distribution or universally quantify over all input distributions. In contrast to these solutions and in addition to the already mentioned focus on runtime systems, the quantitative data flow tracking approach presented in Chapter 8 is independent of any stochastic notion of input data distribution.

In parallel to the work described in Chapter 6, Alvim et al. [13] developed an abstract model for quantitative information flow tracking that accounts for the structure of data. The paper proposes a framework to quantify information leakage independently of the syntactic representation of the secrets, defined in terms of fields that are combined to form structures. The key idea is the notion of "worth assignment" which is introduced to associate each structure with a worth (e.g. in proportion to the harm that would result from disclosure). Like the framework proposed in these pages, [13] distinguishes the notion of data (information, in this case) from the specific representation. However, while promising from a theoretical perspective, no realistic instantiation of the model is discussed in the report. In contrast to the concrete implementations described in Section 3.3 and Section 4.4

and the detailed evaluation in Section 6.5, [13] presents only a theoretical evaluation of the model, based on the comparison with other formal results from information theory.

9.3.2. Dynamic Approaches

Dynamic approaches [19], usually based on dynamic taint analysis, track data flows at runtime, taking into account information which is not available for a static analysis, like user input or the system time, which imposes additional data flow. Note that with the exception of the work described in Chapter 7 (which is a hybrid approach and, as such, discussed in the next subsection) the work presented in this dissertation restricts the standard notion of information flow analysis, which also caters to implicit flows and aims at non-interference assessments [145, 67, 114, 72]: the usage control framework models only flows from container to container. This explains why the term "data flow" is preferred to "information flow".

Several techniques have been proposed for dynamic taint analysis, mainly for detecting malware and unknown vulnerabilities in software [37, 152, 30], checking integrity (tainted data should not affect normal behavior of the program, where tainted means "possibly bad") [37, 38] and confidentiality (public output should not be influenced by tainted data, where tainted stands for "possibly secret") [33, 115, 167].

However, most of them are "hybrid" approaches, because they rely on static annotations to account for implicit flows [42, 155]. In contrast, the general approach behind this work does not require any static annotation to perform the analysis (with the exception, again, of the extension described in Chapter 7). A common pattern in all these solutions is the idea that monitoring should be done "as close to the hardware as possible" [167]. Existing solutions are hence based on binary rewriting [32, 37, 36, 90, 127, 115], memory and pointer analysis [151, 152], partial- or full-system emulation [83, 172, 167, 33, 41] or on making information flow a first-class OS abstraction [48, 169, 96]. In contrast, the intuition behind the multi-layer approach presented in this work, which also motivates the choice of a generic language, is that high-level events such as "print" or "play" or "screenshot" are handled more conveniently at higher layers of abstraction because they can directly been observed there. For this reason, the general model (see Chapter 3 and Chapter 4) is deliberately not bound to any specific architecture or platform.

In [50], a purely dynamic data flow tracking approach is implemented in *TaintDroid* to realize system-wide real-time privacy monitoring in Android, tracking data flows at the variable-, method-, file-, and message-level. Although the results show only a relatively small runtime overhead, TaintDroid's focus is limited to explicit data flow tracking and it is tailored to a specific system (Android). Yin et al. propose *Panorama* [167], a system-wide taint-based data flow tracking approach at the hardware and OS-layer. Panorama generates taint-graphs that represent how data items flow through a system and uses them to detect and identify privacy-breaching malware behavior. Although the approach yields a zero false negatives rate, Panorama slows the system under analysis down by an average factor of 20x and does not cater to implicit flows. [93] presents LibDFT, a solution that uses *shadow tag maps* to store taint marks for each single register and memory address. At runtime these marks are properly propagated (via explicit information flows only) according to the executed binary instruction. Although LibDFT's reported evaluation [93] mentions little performance overhead, these numbers could not be reproduced in the eval-

uation performed in Chapter 7: as the measurements in Table 7.2 show, on all the use cases LibDFT imposes a bigger performance overhead than the solution proposed in Chapter 7. Additionally, LibDFT tracks flows of data only through a certain process, ignoring flows towards OS resources, e.g. files, or other layers of abstraction, e.g. the X11 layer [136], which is instead the goal of this work.

As already mentioned, the goal of all the solutions discussed so far in this section is a dedicated model for data (or, sometimes information-) flow tracking for one specific layer of abstraction. A remarkable exception to this trend stems from the area of provenance aware storage systems: in [122] representations of data are considered at three system layers at the same time (network, file system, workflow engine). Depending on the type of content being handled, the approach relies on different tracking solutions that interacts with each other and exchange taint results across different layers. While the idea of tracking data across different layers of abstraction goes into the same direction of the work presented here, the focus of [122] is not a generic model but rather a three-layer model tailored and customized on the three specific layers discussed in the paper. For example, the taint marks are propagated across the layers without any special form of structured aggregation, in contrast to what can be done with the work described in this dissertation (e.g. in the "Excel saving data to a file"-example see Section 6.1.3).

9.3.3. Hybrid Approaches

Hybrid approaches aim at combining advantages from static and dynamic data flow tracking approaches and to mitigate runtime-overhead in particular. Usually, in a hybrid approach, the application under analysis first undergoes a static analysis phase and after that, based on the results, a dynamic analysis phase.

[31] presents a hybrid, taint-based solution for fine-grained information flow analysis in Java-based applications, with a focus on Java-based server-frontends and client-side browser extensions. Similar to the approach presented in Chapter 7, authors of [31] split their analysis (under the assumption that only Java bytecode is available) into a static and a dynamic analysis phase. During the static phase they compute so called *security annotations* that are used later on during the dynamic phase to track data flow and to enforce security policies by blocking specific method calls at runtime. However, it is unclear how the enforcement actually happens, as denying method calls at the bytecode level may lead to critical instability at runtime. In [144] the authors propose to augment a hybrid data flow tracking approach with declassification rules to downgrade the security levels of specific information flows. This way, all possible flows between input and output channels are detected and enforcement can be performed by allowing, inhibiting, and/or modifying the execution of the intercepted events. Although both [31, 144] show promising results, they do not take into account possible flows between different applications and abstraction layers, missing the system-wide dimension of the problem, which is a major goal of the overall work describe these pages.

[168] proposes another hybrid data flow tracking approach where programmers have to statically specify filter and policy objects that are used at runtime to track the flow of data through an application. [88] and [87] go a step further and decouple information flow tracking logic in a separate thread from application logic. That way, they could reduce the runtime-overhead to the magnitude of 2.3x. Once again, the results in [168, 88, 87] focus

only on one specific abstraction layer and do not take into account inter-process system-wide data flows.

The authors of [41] present *Garm*, which aims at tracking data provenance information across multiple applications and machines. Garm instruments application binaries to track and to store the data flow within and across applications, and beyond that, to monitor interactions with the OS. This solution differs from the general approach presented in this dissertation in many senses: First of all, although it address multiple layers of abstraction at the same time, like [122] it does not describe a general model applicable to different number or type of layers, but rather a "hard-coded" solution for the specific layers of abstraction considered. Secondly, in addition to the more advanced types of policies enforceable by the usage control framework (see Chapter 3), the goal of the provenance tracking that performed in Chapter 8 is fundamentally different from Garm's: the provenance is used to determine *how much* data comes from *where*, whereas Garm, like other approaches in literature, aims at detecting *what* data comes from *where*. Lastly, concerning the instrumentation of applications, although Garm makes also use of a static analysis phase, such task is only a mean to optimize the placement of the runtime monitors (similar, for instance, to [28]); this practically makes Garm a fully-dynamic approach, in contrast to the generic approach describe in this dissertation, which can also integrate static analysis results (see Chapter 7) and provides better performance results[1]

[172] presents *Neon*, a fine-grained data flow tracking approach for derived data management. Neon is implemented as an extension for the XEN Virtual Machine Monitor and tracks data flow on the granularity of individual bytes by tainting each memory address with a *n*-bit taint mark. That taint mark is propagated on each memory write or read access through and across systems. Although Neon presents a sophisticated fine-grained tracking approach to mitigate false positives in the analysis (i.e. data becoming tainted through an unintended dependency), the byte-level tainting approach adopted by Neon limits the number of different data trackable in parallel to 32. In contrast, the system described in Chapter 3 supports a virtually unbounded set of different data items per container. The focus on the byte-level also prevents Neon to capture and properly (i.e. precisely) model the high-level semantics of events, like "taking a screenshot" or "forwarding a mail", which is instead the main strength of the multi-layer approach.

Similar to the extension described in Chapter 7, other approaches in the literature attempted at modeling system-wide information flows, in the sense of inter-application communication, by instrumenting sources and sinks in monitored applications. For the intra-application tracking the literature offers solutions relying on pure dynamic tracking [94] as well as, on static analysis results [141]. All of them, however, perform the inter-application flow tracking relying on the "simultaneous" execution of a sink in the sender application and a source in the receiver. None of them can model a flow towards an external resource, like a file, or toward a non-monitored application. In these scenarios, these approaches lose track of the data, while the solution described in Chapter 7, relying on another monitor at the operating system layer, can cope with these situations, in the latter case by rolling-back to the black-box tracking (see Section 7.3).

[1] The evaluation presented in Section 7.3.4 does not discuss a performance comparison with the Garm tool because, even after contacting the author for the code, it was not possible to get the tool running, mainly due to the legacy libraries and dependencies required. This statement is then supported only by the results described in the original paper [41]

The solution described in Chapter 7 injects the (statically computed) source-sink information into the application byte code using an approach called IRM (**Inline Reference Monitor**) [51]. In the literature, IRM approaches can be found at different layers [171, 156]. IRMs usually inject code into a target applications to intercept sensitive events together with the security property to be enforced and the code to decide its violation. In contrast to this architecture, the solution presented in Chapter 7 injects a reference monitor (PEP, see Section 5.1.1) that notifies events to an external tracker. This allows for tracking data across multiple layers and, possibly, systems in distributed settings and provides the flexibility of changing policies for data and for applications at runtime, without requiring restarting nor re-instrumenting the running applications. In general, any solution based on the architecture discussed in Chapter 5 sees multiple policy enforcement points notifying events to a single (conceptual) centralized decision point, which can then be implemented either a single centralized component or in a distribute fashion [91].

The work described in Chapter 8 was inspired by [115], where authors measure information flow as a network flow capacity. Their tool estimates the amount of information flowing through a program in one (or some) specific execution(s), using static code instrumentation to detect implicit flows. The model described in this pages can be instantiated at different layers of abstraction, including the level of source code considered by McCamant et al, thus generalizing the notion of explicit flows, and does not assume the existence of a control flow, making it suitable also to runtime systems. The main difference between the two approaches is in the interpretation of the analysis results: while estimations provides by the model in Chapter 8 only refer to units of actual data, [115]'s mixes bits of data (due to explicit flows) with bits of indirect information (control flow dependencies), giving a complete different interpretation of the results. Since the usage control framework is based on the observation of events and therefore does not consider control flow dependencies, it does not "quantify" implicit flows, not even when leveraging static analysis results (see Chapter 7). At the cost of manual instrumentation, these are, in contrast, considered in [115].

The idea of measuring information flows by considering information as an incompressible fluid flowing through a network appears also in [161], 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. The work presented in Chapter 8, in contrast, is domain-agnostic and relies only on the size of actions that actually took place, giving a clear interpretation to the results.

10. Conclusions

The work described in this dissertation shows how an existing usage control framework taken from the literature (see Chapter 2) can be augmented with basic data flow tracking capabilities (see Chapter 3) to capture the distinction between data and representations of data. Thanks to its generic nature, the combined model can be instantiated at different layers of abstraction and multiple instances could possibly cooperate for tracking purposes. A detailed description of how to formalize the relation between the layers and how to synchronize their tracking information is presented in Chapter 4, while the architecture of the framework is discussed in Chapter 5.

The expressiveness of policies that can be specified and enforced with this framework is not matched by any other solution from the literature; related works are comparable to this either in terms of information flow control or of usage control (see Chapter 9). As a proof of concept, the model has been instantiated in the context of the reference scenario described in Section 1.4, providing some details about the single layers instantiations in Section 3.3 and about their connection in Section 4.4.

The simplicity of the basic data flow tracking model presented in Chapter 3 introduces coarse overapproximations in concrete instantiations. The second part of this dissertation discusses different strategies to mitigate such imprecision by augmenting the data flow tracking model with additional information about the system.

The first extension (see Chapter 6) leverages information about the *structure* of data; knowing that certain events in the system aggregate data from different sources in a structured manner can help mitigating the overapproximation introduced when data is separated again. The second extension (see Chapter 7) exploits results from *static information flow analysis* at runtime, to model the flows of data through a given application. This offers a sound estimation, potentially more precise than assuming every input flowed to any output and never worse. The third and last extension presented in this work (see Chapter 8), leverages information about the *amount of data* transferred by events in the system. The label creep mitigation, in this case, stems from the additional expressiveness of the language, which allows for specification of policies that apply only to those representations containing at least a certain amount of data.

The decision to present, instantiate and evaluate each of the three extensions individually was motivated by the sake of explanation's simplicity. Each solution showed an improvement in terms of precision, quantified in each evaluation, together with its price in terms of performance loss. In some cases, like in the case of the intra-process data flow tracking, such a tradeoff could be tuned by the choice of different static analyses. Note that the modularity of the combined model allows for a very easy replacement of the data flow tracking component, and the three extensions presented in this work are not intended to be an exhaustive list of alternatives.

Combining extensions The three extensions could also be combined into a single model, provided all the required system information (merge/split events, static analysis results and amount of data transferred by system events) to be available.

Intuitively, such model would look like a "parallel" instantiation of the three solutions: the state of the system would be the cross product of the state of each solution, and every system event would be notified in parallel to each solution. Upon receiving the event, each model would evolve independently from the others. Assuming all the solutions to be sound, the interpretation of the model is the following: At any moment in time, given a container c and a data d, the composed system provides an estimation k of the maximum amount of different units of data d stored in c. How to use this information to express and enforce usage control policies has been already discussed in the first part of this dissertation and in Section 8.4. The value k is given by the estimation provided by the quantitative model for c and d, if according to the other two models d is stored in c; otherwise, $k = 0$.

The intuition is based on the assumption that each solution is sound. If, according to one of the models, data is not stored in a certain container, then any other model that states the opposite does it because of overapproximation, possibly because it is unaware of some aspects of the system, e.g. the existence merge/split events. In other words, the output of the combined model is the "intersection" of the single models' estimations.

Finally, it is worth noticing that the extensions and any combination of them are applicable also when the respective system information is not available, by using fallback values that result in the same precision of the basic model. More specifically, the model for structured data flow tracking can be applied also without any split/merge event in the system; the model for intra-process data flow tracking can be used without static analysis results by replacing them with the black box approach, i.e. mapping every source to every sink; the quantitative model can be applied also if the amount of data transferred by a certain event is not available, because the event can always be conservatively modeled by an edge of infinite size. Nevertheless, the goal of these extensions is to improve the precision by leveraging additional information about the system, and if such information is not available, the lack of precision improvement would not justify the additional performance overhead introduced by any of the different approaches.

Performance The performance results described in this dissertation are measured using non-optimized proof-of-concepts implementations. Additional programming effort is very likely to improve the results, but just by constant factors (the fundamental complexity of some problems, like maximum flow computation, is independent of the concrete implementations). The purpose of these evaluations is to prove the general *feasibility* of the presented ideas, and to investigate which factors affect *scalability* and how.

When arguing about performance, it is worth to remember that the results described in these pages mainly come from *stress-testing* the implementations and that the overhead introduced by the models is *independent of the native execution time* of the monitored events. This means that, for instance, one millisecond required to update the data flow tracking model may be an intolerable overhead when monitoring the execution of a system call, while, at the same time, representing a negligible delay for a BPEL or ESB event [126]. The contribution of this work is not a solution with optimal performance for a specific system or use case, but rather a generic framework for different types of systems and scenarios.

Assumptions The framework for data usage control described in these pages can be effective only if the assumptions listed in Section 1.3 hold. In many scenarios, some of these assumptions may bee too strong or too restrictive, in particular when protecting against motivated attackers in completely open settings. A representative example of the difficulty of this problem is the continuous failures of DRM technologies [57].

Nevertheless, this does not imply that the solutions presented in this work, or, in general, usage control solutions are useless. For instance, while in a completely open environment an assumption like "*the user cannot kill the usage control process*" is indeed unrealistic, in closed or semi-closed environments, like a private corporation's IT infrastructure or a public library's terminal, it is not unreasonable to assume that the user of the system does not have enough privileges to perform such action; similarly, if it is the user himself that runs the framework to prevent the possibility of accidentally misusing the data, it is perfectly reasonable to assume that the user will not kill the framework process, even if he could. Thus, in general, the assumptions behind the work described in these pages can bee too restrictive or absolutely reasonable, depending on the framework's instantiation's context.

Conclusion This research confirmed the initial thesis that it is possible to extend event-driven usage control with data flow tracking concepts. In particular,

- it shows how to formalize a system that supports expression and enforcement of usage control requirements on different representations of the same data at once (Chapter 3), even when data is disseminated across different layers of abstractions (Chapter 4) [Thesis Statement **TS-1**];
- it describes different instantiations of the model (Section 3.3) for at least two different layers of abstraction, confirming its generic nature [Thesis Statement **TS-2**];
- it presents a notion of soundness for data flow tracking (Section 3.4), based on canonical notions from information flow control literature, and proves the soundness of the data flow tracking in terms of such definition (Section 4.2) [Thesis Statement **TS-3**];
- it discusses different ways (Chapter 6, Chapter 7 and Chapter 8) to quantify the precision of the enforcement and to increase it with additional information about the system [Thesis Statement **TS-4**];
- by means of case studies (Section 6.5, Section 7.3 and Section 8.5), it shows empirical evidences, the respective cost in terms of performance overhead, and the situations in which the overhead induced by a more precise model does not compromise the overall system functionality [Thesis Statement **TS-5**].

In conclusion, the work presented in this dissertation addresses the general problem of specifying and enforcing data usage control requirements in a generic system, at and across different layers of abstraction; it discusses how to improve the precision of the enforcement and at what price; and it proves, by mean of case studies, that the ideas are indeed feasible in practice. Depending on the chosen data flow tracking model, some solutions (e.g. structured data flow tracking) may scale better than others for larger numbers of events (e.g. see static analysis, Section 7.3.3 or provenance graphs, Section 8.3.1).

The lesson learned from this work is that it is important to clearly identify the properties of the target system before deciding how to instantiate one of the proposed solutions for data usage control described in these pages. The choice of which approach offers the best tradeoff between precision and performance for the given context is determined, among the rest, by the number, type and frequency of events to be monitored, the additional information about the system and the expressiveness of the requirements to be enforced (i.e. do policies talk about quantity of data?), and the attacker model (system assumptions required to cope with motivated attackers are stronger and may make one solution more effective than another).

For example, consider a system where the main use case involves Alice editing and archiving different reports in compressed files shared with other departments. The structured model may be appropriate if only a monitor at the operating system layer is available and the archiver process is a specific well-defined one; if the source code of the archiver process is available, and is not particularly complex (i.e. static analysis can provide meaningful results in reasonable time), the extension for intra-process data flow tracking may help; if policies talk also about amount of data in a certain representation (see Section 8.4), the quantitative model may be a better choice, as long as the number and frequency of events is low, due to its quadratic complexity. If the number and frequency of events is high, monitoring events at the operating system layer introduce non-negligible overheads; if performance is a priority, maybe a model at the application layer would be more appropriate. However, unless this is combined with an OS-layer one, events like "taking a screenshot of the application's window" may offer a simple solution for a motivated attacker to generate undesired data flows that are not captured by the monitor.

An important result of this work is that no "silver bullet" for data usage control has been found: as extensively argued above and throughout this dissertation, the reason is that a one-size-fit-all solution probably does not even exist, because any approach can be more or less appropriate than another, depending on the properties of the system to control.

10.1. Future Work

While satisfactorily answering the fundamental research question of this thesis, the different parts of the research presented in this dissertation lay the ground for additional future work. First of all, a single model integrating all the different extensions presented in part III is part of the future work. Although each extension introduces minimal changes to the basic model, the integration of multiple extensions introduces some conceptual challenges that require additional work. For instance, quantitatively measuring data structured in different encodings, or quantifying flows of data across layers modeled with static analysis results, require to overcome some fundamental limitations of the model for quantitative data flow tracking presented in Chapter 8 in terms of encoding and compression.

In terms of the general framework, some of the assumptions mentioned in Section 1.3 could be relaxed, e.g. integrating a model for policy evolution [134] to support redistribution of usage controlled data or the analysis of which kind of formal guarantees can the data consumer offers to the data provider during the negotiation phase. Policy management deliberately is not covered by the work presented in this dissertation. Existing work from the literature [99] could be integrated into the usage control framework, possibly

augmenting the architecture with a dedicated component.

The focus of this work was explicitly on isolated physical system; another line of work that can stem from the results presented in this dissertation is the extension of the framework to distributed systems. While some preliminary work based on the the model of Chapter 3 that addresses distributed systems can be found in the literature [91, 92], the additional challenges in this context (see Section 5.1.4) make distributed usage control an interesting source of challenges to tackle.

The models presented in Chapter 3 and Chapter 4 have been formally proved correct (Appendix B) and evaluated in terms of some use cases in the different extensions' evaluation sections. An overall comparison in terms of precision or performance between the models and the extensions presented in the second part would not make much sense due to the fundamental nature and the large number of variables influencing the different approaches. Nevertheless, for a given system in a limited set of use cases, many alternative instantiations of this work are possible, at single or multiple layers of abstraction, and relying on different additional information about the system. For instance, in one instantiation a file could be modeled as a single container, whereas in another one the payload, the filename, and the date of creation could be considered different containers. Similarly, events could propagate sensitivity in a straightforward manner or instead could be carrying information about the structure or the amount of data. The choice of the granularity of the instantiation and of the additional system information integrated into the data flow tracking model directly impacts performance and usability, as proved by the evaluations presented in part II; all these different instantiations could also be compared in a properly designed user study to identify the best tradeoff between security (in terms of precision) and usability (from the user perspective, e.g. in performing certain tasks). Such evaluation would be a good complement to the work described in these pages.

Appendix

A. Data Usage Control Language - Concrete Syntax

The following is an example of concrete syntax, in XML format, for the language described in Chapter 3. The core part is the definition of a a mechanism (line 197 - line 206), reported here for ease of explanation:

```
197 <complexType name="MechanismBaseType">
198   <sequence>
199     <element name="timestep" type="tns:TimeAmountType" minOccurs="0" maxOccurs="1" />
200     <element name="trigger" type="tns:EventMatchingOperatorType" minOccurs="0"
      maxOccurs="1" />
201     <element name="condition" type="tns:ConditionType" minOccurs="0" maxOccurs="1" />
202     <element name="authorizationAction" type="tns:AuthorizationActionType"
      minOccurs="0" maxOccurs="0" />
203     <element name="executeAction" type="tns:ExecuteActionType" minOccurs="0"
      maxOccurs="unbounded" />
204   </sequence>
205   <attribute name="name" type="string" use="required" />
206 </complexType>
```

Each mechanism is uniquely identified by a name (line 205). The first (optional) element of a mechanism is called timestep (line 199) and specifies the amount of time within which events are considered to be happening at the same time. If unspecified, this corresponds to a single timestep of the monitor, meaning that no event can be happening "at the same time" of another event. This is an obvious consequence of processing events in a serialized fashion (see Section 2.1.3), and the purpose of this field is to overcome such limitation and define the size of a timestep for each single mechanism.

The trigger element (line 200) corresponds to the trigger event of an ECA mechanism, while line 201 contains the condition. Note that both are optional argument of a mechanism: it the event is not specified, the mechanism will be triggered at the end of each timestep (as defined above), whereas an empty condition corresponds to the default value of *true*. The action part of an ECA mechanism is modeled by two elements: an authorization action (line 202), which specifies whether the trigger event is allowed to be executed or not (line 189-line 195), possibly with modified parameters (line 186), and a list of additional actions to execute (line 203). With the different combinations of these two elements it is possible to model all the different types of ILPs described in Section 2.3.

The rest of the specification covers aspects like the definition of time ranges (line 3-line 26), the specification of conditions, with the list of all the operators (line 41-line 174), and the definition of a policy as a list of mechanisms (line 208-line 215). Note that state-based operators (line 124-line 129) are modeled with a single operator that takes one mandatory parameter and two optional additional parameters. The syntax check, in this case, is performed within the code that implements the PIP. In addition to the operators specified in Chapter 3), this choice is motivated by the additional offered flexibility of

embodying new data-flow-related operators, like e.g. the quantitative measurements operators defined in Section 8.4, without the need to modify the language specification.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <schema xmlns="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www22.in.tum.de/enforcementLanguage"
   xmlns:tns="http://www22.in.tum.de/enforcementLanguage"
   xmlns:event="http://www22.in.tum.de/enforcementLanguage/event"
   xmlns:cnd="http://www22.in.tum.de/enforcementLanguage/condition"
   elementFormDefault="qualified">
3    <simpleType name="TimeUnitType">
4      <restriction base="string">
5        <enumeration value="TIMESTEPS" />
6        <enumeration value="NANOSECONDS" />
7        <enumeration value="MICROSECONDS" />
8        <enumeration value="MILLISECONDS" />
9        <enumeration value="SECONDS" />
10       <enumeration value="MINUTES" />
11       <enumeration value="HOURS" />
12       <enumeration value="DAYS" />
13       <enumeration value="WEEKS" />
14       <enumeration value="MONTHS" />
15       <enumeration value="YEARS" />
16     </restriction>
17   </simpleType>
18
19   <attributeGroup name="TimeAmountAttributeGroup">
20     <attribute name="amount" type="long" use="required" />
21     <attribute name="unit" type="tns:TimeUnitType" use="optional" default="TIMESTEPS"
       />
22   </attributeGroup>
23
24   <complexType name="TimeAmountType">
25     <attributeGroup ref="tns:TimeAmountAttributeGroup" />
26   </complexType>
27
28   <complexType name="ParameterType">
29     <attribute name="name" type="string" use="required" />
30     <attribute name="value" type="string" use="required" />
31   </complexType>
32
33   <complexType name="ExecuteActionType">
34     <sequence>
35       <element name="parameter" type="tns:ParameterType" minOccurs="0"
         maxOccurs="unbounded" />
36     </sequence>
37     <attribute name="name" type="string" use="required" />
38     <attribute name="id" type="string"></attribute>
39   </complexType>
40
41   <complexType name="ConditionType">
42     <sequence>
43       <group ref="tns:Operators" minOccurs="1" maxOccurs="1" />
44     </sequence>
45   </complexType>
46   <complexType name="ConditionParamMatchType">
47     <attribute name="name" type="string" use="required" />
48     <attribute name="value" type="string" use="required" />
49   </complexType>
50   <complexType name="NotType">
51     <sequence>
52       <group ref="tns:Operators" />
53     </sequence>
54   </complexType>
55   <complexType name="TrueType">
```

```
56      <sequence />
57    </complexType>
58    <complexType name="FalseType">
59      <sequence />
60    </complexType>
61    <complexType name="OrType">
62      <sequence>
63        <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
64      </sequence>
65    </complexType>
66    <complexType name="AndType">
67      <sequence>
68        <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
69      </sequence>
70    </complexType>
71    <complexType name="ImpliesType">
72      <sequence>
73        <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
74      </sequence>
75    </complexType>
76    <complexType name="SinceType">
77      <sequence>
78        <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
79      </sequence>
80    </complexType>
81    <complexType name="AlwaysType">
82      <sequence>
83        <group ref="tns:Operators" />
84      </sequence>
85    </complexType>
86    <complexType name="BeforeType">
87      <sequence>
88        <group ref="tns:Operators" />
89      </sequence>
90      <attributeGroup ref="tns:TimeAmountAttributeGroup" />
91    </complexType>
92    <complexType name="DuringType">
93      <sequence>
94        <group ref="tns:Operators" />
95      </sequence>
96      <attributeGroup ref="tns:TimeAmountAttributeGroup" />
97    </complexType>
98    <complexType name="WithinType">
99      <sequence>
100       <group ref="tns:Operators" />
101     </sequence>
102     <attributeGroup ref="tns:TimeAmountAttributeGroup" />
103   </complexType>
104   <complexType name="RepLimType">
105     <sequence>
106       <group ref="tns:Operators" />
107     </sequence>
108     <attributeGroup ref="tns:TimeAmountAttributeGroup" />
109     <attribute name="lowerLimit" type="long" use="required" />
110     <attribute name="upperLimit" type="long" use="required" />
111   </complexType>
112   <complexType name="RepSinceType">
113     <sequence>
114       <group ref="tns:Operators" minOccurs="2" maxOccurs="2" />
115     </sequence>
116     <attribute name="limit" type="long" use="required" />
117   </complexType>
118   <complexType name="RepMaxType">
119     <sequence>
120       <group ref="tns:Operators" />
```

```
121     </sequence>
122     <attribute name="limit" type="long" use="required" />
123   </complexType>
124   <complexType name="StateBasedOperatorType">
125     <attribute name="operator" type="string" use="required" />
126     <attribute name="param1" type="string" use="required" />
127     <attribute name="param2" type="string" use="optional" />
128     <attribute name="param3" type="string" use="optional" />
129   </complexType>
130   <complexType name="EvalOperatorType">
131     <sequence>
132       <element name="content" type="string" />
133     </sequence>
134     <attribute name="type" type="string" use="required" />
135   </complexType>
136   <group name="Operators">
137     <choice>
138       <element name="true" type="tns:TrueType" />
139       <element name="false" type="tns:FalseType" />
140       <element name="not" type="tns:NotType" />
141       <element name="or" type="tns:OrType" />
142       <element name="and" type="tns:AndType" />
143       <element name="implies" type="tns:ImpliesType" />
144       <element name="eventMatch" type="tns:EventMatchingOperatorType" />
145       <element name="conditionParamMatch" type="tns:ConditionParamMatchType" />
146       <element name="since" type="tns:SinceType" />
147       <element name="always" type="tns:AlwaysType" />
148       <element name="before" type="tns:BeforeType" />
149       <element name="during" type="tns:DuringType" />
150       <element name="within" type="tns:WithinType" />
151       <element name="repLim" type="tns:RepLimType" />
152       <element name="repSince" type="tns:RepSinceType" />
153       <element name="repMax" type="tns:RepMaxType" />
154       <element name="stateBasedFormula" type="tns:StateBasedOperatorType" />
155       <element name="eval" type="tns:EvalOperatorType" />
156     </choice>
157   </group>
158   <simpleType name="ParamMatchDataTypes">
159     <restriction base="string">
160       <pattern value="dataUsage|containerUsage" />
161     </restriction>
162   </simpleType>
163   <complexType name="ParamMatchType">
164     <attribute name="name" type="string" use="required" />
165     <attribute name="value" type="string" use="required" />
166     <attribute name="type" type="tns:ParamMatchDataTypes" use="optional"
      default="containerUsage" />
167   </complexType>
168   <complexType name="EventMatchingOperatorType">
169     <sequence>
170       <element name="paramMatch" type="tns:ParamMatchType" minOccurs="0"
        maxOccurs="unbounded" />
171     </sequence>
172     <attribute name="action" type="string" use="required" />
173     <attribute name="isActual" type="boolean" use="required" />
174   </complexType>
175
176   <complexType name="ModifyActionType">
177     <sequence>
178       <element name="parameter" type="tns:ParameterType" minOccurs="0"
        maxOccurs="unbounded" />
179     </sequence>
180   </complexType>
181   <complexType name="AuthorizationInhibitType">
182     <sequence />
```

```
183   </complexType>
184   <complexType name="AuthorizationAllowType">
185     <sequence>
186       <element name="modify" type="tns:ModifyActionType" minOccurs="0" maxOccurs="1" />
187     </sequence>
188   </complexType>
189   <complexType name="AuthorizationActionType">
190     <choice>
191       <element name="allow" type="tns:AuthorizationAllowType" />
192       <element name="inhibit" type="tns:AuthorizationInhibitType" />
193     </choice>
194     <attribute name="name" type="string" use="required" />
195   </complexType>
196
197   <complexType name="MechanismBaseType">
198     <sequence>
199       <element name="timestep" type="tns:TimeAmountType" minOccurs="0" maxOccurs="1" />
200       <element name="trigger" type="tns:EventMatchingOperatorType" minOccurs="0"
        maxOccurs="1" />
201       <element name="condition" type="tns:ConditionType" minOccurs="0" maxOccurs="1" />
202       <element name="authorizationAction" type="tns:AuthorizationActionType"
        minOccurs="0" maxOccurs="0" />
203       <element name="executeAction" type="tns:ExecuteActionType" minOccurs="0"
        maxOccurs="unbounded" />
204     </sequence>
205     <attribute name="name" type="string" use="required" />
206   </complexType>
207
208   <complexType name="PolicyType">
209     <sequence>
210       <element name="mechanism" type="tns:MechanismBaseType" minOccurs="1"
        maxOccurs="unbounded" />
211     </sequence>
212     <attribute name="name" type="string" use="required" />
213   </complexType>
214
215   <element name="policy" type="tns:PolicyType" />
216
217 </schema>
```

B. Soundness Proof - Cross-layer

B.1. Soundness of $\hat{\mathcal{R}}_{A\otimes B}$

Proof. Let $\sigma = (\sigma_A, \sigma_B)$ be the initial state of the system, and σ'_A the state at layer A after executing the trace (t_A, t_B) ($\sigma'_A = \hat{\mathcal{R}}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))|_A$). Let $t_{A\otimes B} = \gamma_A(t_A) \bowtie \gamma_B(t_B)$, and let $\sigma_\perp$ be the initial state at bottom such that $\sigma_\perp \vdash \sigma_A$. Finally, let $\sigma'_\perp = \mathcal{R}^{\#}_{\perp}(\sigma_\perp, t_{A\otimes B})$. If $\hat{\mathcal{R}}_{A\otimes B}$ is sound, then σ'_A is sound w.r.t $\sigma'_\perp$ ($\sigma'_\perp \vdash \sigma'_A$).

Assume that σ'_A is not sound.

This implies that there exists a data item d and a container c_A such that $d \notin \sigma'_A(c_A)$ but $\exists\, c_\perp \in \sigma'_\perp : d \in \sigma'_\perp(c_\perp) \wedge c_\perp \in \gamma(c_A)$.

By definition of $\mathbb{S}_{\mathcal{R}^{\#}_{\perp}}$ then either (1) $d \in \sigma_\perp(c_\perp)$ or (2) d has been transferred to $c_\perp$ by events in $t_{A\otimes B}$, i.e. $\exists\, c'_\perp \in \mathbb{S}_{\mathcal{R}^{\#}_{\perp}}(t_{A\otimes B}) : d \in \sigma_\perp(c'_\perp)$ and $c_\perp \in \mathbb{D}_{\mathcal{R}^{\#}_{\perp}}(t_{A\otimes B})$.

(1) cannot be the case, since $\sigma_\perp \vdash \sigma_A$ implies $d \in \sigma_A(c_A)$ and the composed monitor only adds data to containers ($d \in \sigma_A(c_A) \implies d \in \sigma'_A(c_A)$).

Therefore (2) must hold. By soundness of $\mathcal{R}_A$ and $\mathcal{R}_B$, it follows that $c'_\perp \in \mathbb{S}_{\mathcal{R}^{\#}_{\perp}}(t_{A\otimes B}) \subseteq \gamma_A(\mathbb{S}_{\mathcal{R}_A}(t_A)) \cup \gamma_B(\mathbb{S}_{\mathcal{R}_B}(t_B))$ and $c_\perp \in \mathbb{D}_{\mathcal{R}^{\#}_{\perp}}(t_{A\otimes B}) \subseteq \gamma_A(\mathbb{D}_{\mathcal{R}_A}(t_A)) \cup \gamma_B(\mathbb{D}_{\mathcal{R}_B}(t_B))$. Thus, the abstraction of $c'\perp$ and $c_\perp$ must belong to the sources and destinations of the trace at the higher level, i.e. $\exists\, c_{A\otimes B} \in \mathbb{S}_{A\otimes B} : c_{A\otimes B} \in \alpha_{A\otimes B}(c_\perp) \wedge d \in \sigma(c_{A\otimes B})$ and $\exists\, c'_{A\otimes B} \in \mathbb{D}_{A\otimes B} : c'_{A\otimes B} \in \alpha_{A\otimes B}(c_\perp)$. This implies that either $c_A = c'_{A\otimes B}$ or $c_A \sim c'_{A\otimes B}$. But then, since $\hat{\mathcal{R}}_{A\otimes B}$ propagates all data in $\mathbb{S}_{A\otimes B}$ to $\mathbb{D}_{A\otimes B}$ and to all related containers, then $d \in c_A$, which is a contradiction. A similar argument holds for an unsound approximation of B.

□

Figure B.1 depicts the partitions of the memory in sources/non-sources and destinations/non-destinations according to monitoring at each layer. Recalling the property of source and destination sets (autorefsec:ucdft:sourceDest) there cannot be flows to memory regions outside the destination set.

B.2. Soundness of $\dot{\mathcal{R}}_{A\otimes B}$

Proof. In this proof, an inductive argument for the soundness of the approach over the length of the trace is provided.

Base Case. Given a trace composed by the single event e, and assuming the state of the system is the initial state σ^i, there are three cases for the cross-layer behavior of e, and they are defined by the oracle X_B (Algorithm 2). Assume e is an event in A (the case for B is analogous).

If the behavior of e is `INTRA`, it means that e is not part of any cross-layer flow and is thus modeled as a layer internal event using $\mathcal{R}^{-}_{A}$. In this case, the execution of e may

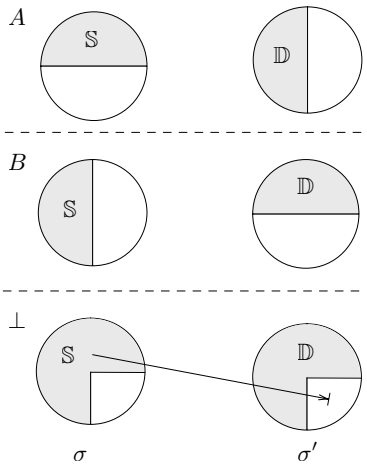


Figure B.1.: Composition of potential flows between layers for a trace (t_A, t_B). Circles represent the whole memory. Partitions are induced by sources and destinations of the trace at the respective layer. Note the resulting Non-Interference property at $\perp$ (arrow).

either generate new flows of data within A, (captured by $\mathcal{R}_A^-$ and sound because $\mathcal{R}_\perp \vdash \mathcal{R}_A^-$), or to B (via related containers). The latter kind of flows is captured by the *sync* operation (line 22), which propagates the possibly new content to the related containers in B, identified by the X_A oracle.

If the behavior of the first event is `OUT`, the algorithm will write the content of its sources to its target (line 17) (sound estimation thanks to single layer soundness) and to an intermediate container (line 16). Because intermediate containers have no concretization in $\perp$, their content is irrelevant for soundness checks; the resulting state at $\perp$ is then the same as for the `INTRA` case and, after syncing the content with the containers at the other layer (line 22), the soundness can be confirmed using the same argument.

The case of e being an `IN` event is not considered because the first event cannot be an `IN` event by assumption (traces represent real executions, and `IN` events only take place after at least one respective `OUT` event, see Section 4.3.3).

Inductive Case. Given $t = t^p || \langle e \rangle$ and assuming $\sigma = \dot{\mathcal{R}}_{A\otimes B}(\sigma^i_{A\otimes B}, t^p)$ sound w.r.t $\sigma_\perp = \mathcal{R}_\perp(\sigma^i_\perp, \gamma_{A\otimes B}(t^p))$ by inductive hypothesis, the goal of this step is to show that state $\sigma' = \dot{\mathcal{R}}_{A\otimes B}(\sigma, e)$ is also sound. As before, e is assumed to be an event at layer A (the case for B is analogous).

As in the previous case, e can be either an `INTRA`, an `IN` or an `OUT` event.

If e is an `INTRA` event, then the same argument of the base case applies, i.e. the estimation given by $\mathcal{R}_A^-$ and the *sync* operator is sound.

If e is an `OUT` event, then the behavior remains the same as for the base case: `OUT` events are equivalent to `INTRA` event because the intermediate containers are not relevant for soundness purposes due to their empty $\perp$-concretization.

The core of the model is the transition relation in case e is an `IN` event. Let sc be the scope of e given by X_B. Every container in the destination of e is updated with the data

stored in the intermediate container c_{sc} (line 11).

The intuition of soundness of this step is the following: Let E^O be the set of all those OUT event in t^p associated to the same scope sc of e. Each event in E^O transferred the content of its sources to c_{sc} (line 16). The content of such sources is a sound overapproximation of the content of their concretization, because any state reached during execution of trace t^p is sound by inductive hypothesis. Let t_{e^o} be the subtrace of t from the beginning until e^o (excluded).

Because no event can delete content from c_{sc}, $\sigma(c_{sc})$ is a conservative estimation of the content of all the sources of the events in E^O, i.e.

$$\sigma(c_{sc}) \supseteq \bigcup_{e^o \in E^O} \bigcup_{c \in \mathbb{S}_{\mathcal{R}^{\#}_{\perp}}(\gamma_{A\otimes B}(e^o))} \mathcal{R}^{\#}_{\perp}(\gamma_{A\otimes B}(\sigma), \gamma_{A\otimes B}(t_{e_o}))(c),$$

which by Definition 4.12 is a superset of the content of the concretization of the destination containers of e. Thus, transferring the content of $\sigma(c_{sc})$ to the destinations of e (line 11) results in a sound state.

In more details, it is clear from the algorithm (line 11) that e only appends the content of the intermediate container to its target. If this violates the soundness definition, there must exists a container $c \in \mathbb{D}_{\mathcal{R}^-_A}(e)$ and a data $d \in \mathcal{D}$ such that $d \notin \sigma'(c)$ while there exists a container $c_\perp \in \gamma_{A\otimes B}(c)$ that contains d in $\sigma'_\perp = \mathcal{R}_\perp(\sigma_\perp, \gamma_A(e))$. Note that:

1. If $d \in \sigma_\perp(c_\perp)$, the soundness of σ requires that $d \in \sigma(c)$. But $d \in \sigma(c)$ is not possible, because IN events such as e only append data to their destination, and $d \in \sigma(c) \implies d \in \sigma'(c)$. Therefore $d \notin \sigma(c_\perp)$.

2. If $d \notin \sigma(c_\perp)$ and $d \in \sigma'(c_\perp)$, it means that the event $e_\perp \in \mathcal{E}_\perp$ that transferred d to $c_\perp$ must be part of $\gamma_A(e)$.

3. If $e_\perp \in \gamma_A(e)$, $d \in \mathcal{R}^-_A(\sigma, e)(c)$, because $\mathcal{R}^-_A$ is sound. But $d \notin \dot{\mathcal{R}}_{A\otimes B}(\sigma, e)(c)$. Considering that e is an IN event, this can only happen if $d \in \sigma(c^{sc})$, where sc is the scope of the cross-layer event e is part of (line 11). In order for $\dot{\mathcal{R}}_{A\otimes B}$ to be unsound, it must be the case that $d \notin \sigma(c^{sc})$.

4. Because e is an IN event, at least one corresponding OUT event must have taken place in t^p. Let E^O be the set of all the OUT events in t^p that are related to e (i.e. with the same scope id sc). From Definition 4.12, there must exists an event $e^o \in E^O$ in the trace, such that $d \in \sigma^O_\perp(\gamma_B(\mathbb{S}_{\mathcal{R}^-_B}(e^O)))$, where σ^O is the state of the system right before the execution of e^O and $\sigma^O_\perp$ is its concretization.

5. Considering that σ^O is a state of the system reachable by a subtrace of t, $\sigma^O_\perp \vdash \sigma^O$ by inductive hypothesis. Such soundness implies that if $d \in \sigma^O_\perp(\gamma_B(\mathbb{S}_{\mathcal{R}^-_B}(e^O)))$, then $d \in \sigma^O(\mathbb{S}_{\mathcal{R}^-_B}(e^O))$. Being e^O an OUT event with respect to scope sc, $d \in \dot{\mathcal{R}}_{A\otimes B}(\sigma^O, e^O)(s^{sc})$ (line 16). In particular, because the content of intermediate containers is never erased (multiple repetitions of the same cross action are assigned different scope ids), $d \in \dot{\mathcal{R}}_{A\otimes B}(\sigma^O, e^O)(s^{sc}) \implies d \in \sigma(s^{sc})$, which is impossible because $d \notin \sigma(c^{sc})$ (see point 3)

6. $\implies$ absurd, i.e. it is not possible that the state $\sigma' = \dot{\mathcal{R}}_{A\otimes B}(\sigma, e)$ is not sound.

□ □

C. Soundness Proof - Quantitative Data Flow Tracking

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 Chapter 8, this would have obscured the dual nature of edge labels as capacities and actual flows — actual flows are never infinite.

C.1. Correctness of Tracking

To show correctness of the model in the sense of Section 8.3, $\forall t \in \mathbb{N} \forall c \in C : \varphi(c, t) \leq \kappa((c, t))$ must be proved.

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

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 provenance graph 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, i.e. $\varphi(c, t) \leq \kappa((c, t))$.

If, instead, $m < \kappa((c, t'))$, then node (c, t) is added as a successor of (c, t') with an edge labelled 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, i.e. $\varphi(c, t) \leq \kappa((c, t))$.

C1 step: ℓ units of data are to be copied from c_A to a c_B that does not exist at time $t-1$. This introduces 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, it holds that $\varphi(c_B, t) \leq \min(\ell, \varphi(c_A, t'))$. 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 $\kappa((c_B, t)) = \ell$. 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 a copy of ℓ units of data from container c_A to the existing container c_B. By construction of the provenance graph, 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 $e_k = \big((c_B, t''), \infty, (c_B, t)\big) \in \mathcal{E}_t$.

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. These edge labels can be interpreted as upper bounds for the flow of *sensitive* units: For each edge (action), it is impossible that more sensitive than overall units flow.

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 provenance graph 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 $\varphi(c_B, t)$. □

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.

C.2. Correctness of Optimizations

This section proves equivalence $\sim$ of a graph $\mathcal{G}_t$ with its optimized version $\mathcal{G}'_t$ according to the rules presented in Section 8.3.

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

$$
\begin{aligned}
\exists k \in \mathbb{N} \forall c_a, c_b, c \in C, q, x \in \mathbb{N}, i \in [k+1..t-1], j \in [0..k-1] : \\
\varepsilon_k = C(c_a, q, c_b) \wedge \varepsilon_t = T(c_b, m) \wedge \\
\varepsilon_j \neq I(c_b, x) \wedge \varepsilon_j \neq C(c, x, c_b) \wedge \\
\varepsilon_i \neq T(c_b, x) \wedge \varepsilon_i \neq C(c, x, c_b) \wedge \\
\kappa((c_b, k)) > m \wedge s_Y + m \leq q
\end{aligned}
$$

where $s_Y = \sum_{\varepsilon_i = C(c_b,x,c) \in tr} x$, let tr' be the 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 $\varepsilon_i = C(c_b, q_i, c_i)$ and $\varepsilon'_i = \varepsilon_i$ otherwise, for every $i \in [k+1..t-1]$. The following shows that tr' results in a simpler (i.e. *smaller*), yet equivalent graph than tr.

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 $t-1$, let (c_b, k) be the source node of the truncation action at time t. Let

$$\begin{aligned}
&\mathcal{N}'_t = \mathcal{N}_t \setminus \{(c_b, t)\} \qquad \text{and} \\
&\mathcal{E}'_t = \Big(\mathcal{E}_t \setminus \big(\{((c_b,k), m, (c_b,t)), ((c_a,u), q, (c_b,k))\} \cup \textstyle\bigcup_{\varepsilon_i = C(c_b,q_i,c_i)} \{((c_b,k), q_i, (c_i,i))\}\big)\Big) \cup \\
&\qquad \big(\{((c_a,u), m, (c_b,k))\} \cup \textstyle\bigcup_{\varepsilon_i = C(c_b,q_i,c_i)} \{((c_a,u), q_i, (c_i,i))\}\big).
\end{aligned}$$

Let $X \subseteq C$ be a set of containers. $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d) \implies ((c_a,u), q, (c_b,k)) \in mincut_{\mathcal{G}_{t,X}}((S,0), d)$ 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

$$\begin{aligned}
\exists k \in \mathbb{N} \forall c, c_a \in C, q \in \mathbb{N}, i \in [k+1..t-1], x \in \mathbb{N} : \\
\varepsilon_k = C(c_a, s_1, c_b) \wedge \varepsilon_t = C(c_c, s_2, c_b) \wedge \\
\varepsilon_i \neq T(c_b, x) \wedge \varepsilon_i \neq C(c, x, c_b) \wedge \\
\varepsilon_i \neq C(c_b, x, c) \wedge \varepsilon_i \neq T(c_a, x) \wedge \varepsilon_i \neq C(c, x, c_a),
\end{aligned}$$

let tr' be the new sequence $tr' = (\varepsilon_0, \ldots, \varepsilon_{k-1}, C(c_a, s_1, c_c, s_2, c_b), \ldots, \varepsilon_{t-1})$. tr' 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. Let

$$\begin{aligned}
&\mathcal{N}'_t = \mathcal{N}_t \setminus \{(c_b, t)\} \qquad \text{and} \\
&\mathcal{E}'_t = (\mathcal{E}_t \setminus \{((c_b,k), \infty, (c_b,t)), ((c_c,w), s_2, (c_b,t))\}) \cup \{((c_c,w), s_2, (c_b,k))\}.
\end{aligned}$$

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 $c_b \in X$,

$$\begin{aligned}
\kappa_{t,X}(d) \neq \kappa'_{t,X}(d) \implies \big(&((c_c,w), s_2, (c_b,k)) \in mincut_{\mathcal{G'}_{t,X}}((S,0), d) \wedge \\
&((c_c,w), s_2, (c_b,t)) \notin mincut_{\mathcal{G}_{t,X}}((S,0), d)\big).
\end{aligned}$$

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

$$\begin{aligned}
\exists k \in \mathbb{N} \forall c_a, c \in C, q, x \in \mathbb{N}, i \in [k+1..t-1], j \in [0..k-1] : \\
\varepsilon_k = C(c_a, s_1, c_b) \wedge \varepsilon_t = C(c_a, s_2, c_b) \wedge \\
\varepsilon_j \neq I(c_b, x) \wedge \varepsilon_j \neq C(c, x, c_b) \wedge \\
\varepsilon_i \neq T(c_b, x) \wedge \varepsilon_i \neq C(c, x, c_b) \wedge \\
\varepsilon_i \neq T(c_a, x) \wedge \varepsilon_i \neq C(c, x, c_a) \wedge \\
s_1 > \textstyle\sum_{\varepsilon_i = C(c_b,x,c)} x
\end{aligned}$$

let tr' be the new sequence $tr' = (\varepsilon_0, \ldots, \varepsilon_{k-1}, C(c_a, s_1 + s_2, c_b), \ldots, \varepsilon_{t-1})$. tr' results in a simpler, yet equivalent graph.

Let (c_a, u) be the source node of the copy action at time k and

$$Y = \{((c_b, k), q_i, (c_i, i)) | i \in [k+1..t-1], c_i \in \mathcal{C}, q_i \in \mathbb{N}, \varepsilon_i = C(c_b, q_i, c_i) \in tr\}$$

the destination nodes of the copy steps from c_b between time $k+1$ and time $t-1$. Let s_Y be the sum of those edges' capacities. Let

$$\begin{aligned} \mathcal{N}'_t &= \mathcal{N}_t \backslash \{(c_b, t)\} \qquad \text{and} \\ \mathcal{E}'_t &= \mathcal{E}_t \backslash \{((c_b, k), \infty, (c_b, t)), ((c_a, u), s_1, (c_b, k)), ((c_a, u), s_2, (c_b, t))\} \\ &\quad \cup \{((c_a, u), s_1 + s_2, (c_b, k))\} \end{aligned}$$

Let $X \subseteq C$ be a set of containers. $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d) \implies Y \cap mincut_{\mathcal{G}_{t,X}}((S, 0), d) \neq \emptyset$. Let s_X be the sum of those edges' capacities, i.e.

$$s_X = \sum_{((c_b,k),q_i,(c_i,i)) \in Y \cap mincut_{\mathcal{G}_{t,X}}((S,0),d)} q_i.$$

Since $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d)$, it means $s_1 < s_X$. But this is impossible since $s_X \leq s_Y \leq s_1$. Hence $\mathcal{G}'_t \sim \mathcal{G}_t$.

Bibliography

[1] An Act To amend the Internal Revenue Code of 1986 to improve portability and continuity of health insurance coverage in the group and individual markets, to combat waste, fraud, and abuse in health insurance and health care delivery, to promote the use of medical savings accounts, to improve access to long-term care services and coverage, to simplify the administration of health insurance, and for other purposes. http://www.gpo.gov/fdsys/pkg/CRPT-104hrpt736/pdf/CRPT-104hrpt736.pdf(last access: 11.05.2015).

[2] An Act to protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes. http://www.gpo.gov/fdsys/pkg/PLAW-107publ204/pdf/PLAW-107publ204.pdf(last access: 11.05.2015).

[3] Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. http://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:31995L0046(last access: 11.05.2015).

[4] Java Grande Forum Benchmark Suite. https://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html(last access: 11.05.2015).

[5] JavaFTP, http://sourceforge.net/projects/javaftp/ (last access: 16.06.2014.).

[6] JOANA. http://joana.ipd.kit.edu(last access: 11.05.2015).

[7] Linux Mint, http://www.linuxmint.com/(last access: 11.05.2015).

[8] OW2-ASM instrumentation framework. http://asm.ow2.org/(last access: 11.05.2015).

[9] Strace, http://sourceforge.net/projects/strace/. Last Access: 11.05.2015.

[10] T.J.Watson Library for Analysis (WALA), http://wala.sf.net(last access: 11.05.2015).

[11] Multimedia framework (MPEG-21) – Part 5: Rights Expression Language, 2004. ISO/IEC standard 21000-5:2004.

[12] Adobe. Adobe LiveCycle Rights Management ES. `http://www.adobe.com/products/livecycle/modules.displayTab3.html`(last access: 11.05.2015), August 2010.

[13] Mário S. Alvim, Andre Scedrov, and Fred B. Schneider. When not all bits are equal: Worth-based information flow. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 120–139. Springer Berlin Heidelberg, 2014.

[14] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 91–102, New York, NY, USA, 2006. ACM.

[15] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer Berlin Heidelberg, 2004.

[16] G.F. Anastasi, E. Carlini, M. Coppola, P. Dazzi, A. Lazouski, F. Martinelli, G. Mancini, and P. Mori. Usage control in cloud federations. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 141–146, March 2014.

[17] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[18] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). IBM Technical Report, 2003. `http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/`(last access: 11.05.2015).

[19] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM.

[20] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for android. In Sushil Jajodia and Jianying Zhou, editors, *Security and Privacy in Communication Networks*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 326–343. Springer Berlin Heidelberg, 2010.

[21] H. Balinsky, D.S. Perez, and S.J. Simske. System call interception framework for data leak prevention. In *Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International*, pages 139–148, Aug 2011.

[22] Jean-Pierre Banâtre, Ciarán Bryce, and Daniel Le Métayer. Compile-time detection of information flow in sequential programs. In *Proceedings of the Third European Symposium on Research in Computer Security*, ESORICS '94, pages 55–73, London, UK, UK, 1994. Springer-Verlag.

[23] David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. Monpoly: Monitoring usage-control policies. In *2nd International Conference on Runtime Verification (RV 2011)*. LNCS, 2011.

[24] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *15th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 23–33. ACM Press, 2010.

[25] Andreas Bauer, Martin Leucker, and Jonathan Streit. SALT — Structured Assertion Language for Temporal Logic. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 757–775, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] Pascal Birnstill and Alexander Pretschner. Enforcing privacy through usage-controlled video surveillance. In *10th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 318–323, Aug 2013.

[27] Asish Kumar Biswas. Towards improving data driven usage control precision with intra-process data flow tracking. Master's thesis, Department of Informatics, Technische Universität München, 2014.

[28] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg, 2010.

[29] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[30] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*, pages 463–475, Dec 2007.

[32] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, volume Volume 13, pages 22–22. USENIX, USENIX Association, 2004.

[34] D Clark, S Hunt, and P Malacaria. Quantitative Analysis of the Leakage of Confidential Data. *Electronic Notes in Theoretical Computer Science*, 59:238–251, 2002.

[35] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Belief in information flow. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, CSFW '05, pages 31–45, Washington, DC, USA, 2005. IEEE Computer Society.

[36] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[37] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Transactions on Computer Systems (TOCS)*, 26(4):9:1–9:68, December 2008.

[38] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society.

[39] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 546–569, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In Morris Sloman, EmilC. Lupu, and Jorge Lobo, editors, *Policies for Distributed Systems and Networks*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer Berlin Heidelberg, 2001.

[41] Brian Demsky. Garm: cross application data provenance and policy enforcement. In *Proceedings of the 4th USENIX conference on Hot topics in security*, HotSec'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[42] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[43] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[44] Robling Denning and Dorothy Elizabeth. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[45] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET Run Time Monitor: Tool Demonstration. *Electronic Notes in Theoretical Computer Science*, 253(5):153–159, 2009.

[46] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 297–312, Washington, DC, USA, 2011. IEEE Computer Society.

[47] R. Iannella (ed.). Open Digital Rights Language v1.1, 2008. `http://odrl.net/1.1/ODRL-11.pdf`(last access: 11.05.2015).

[48] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17–30, October 2005.

[49] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[50] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[51] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004.

[52] Úlfar Erlingsson and Fred B. Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, pages 87–95, New York, NY, USA, 2000. ACM.

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

[54] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[55] Denis Feth and Alexander Pretschner. Flexible data-driven security for android. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, SERE '12, pages 41–50, Washington, DC, USA, 2012. IEEE Computer Society.

[56] Apache Software Foundation. Apache Thrift software framework, `https://thrift.apache.org`(last access: 11.05.2015).

[57] Electronic Frontier Foundation. Digital Rights Management: A failure in the developed world, a danger to the developing world. `https://www.eff.org/wp/digital-rights-management-failure-developed-world-danger-developing-world`(last access: 11.05.2015).

[58] Alexander Fromm, Florian Kelbert, and Alexander Pretschner. Data protection in a cloud-enabled smart grid. In Jorge Cuellar, editor, *Smart Grid Security*, volume 7823 of *Lecture Notes in Computer Science*, pages 96–107. Springer Berlin Heidelberg, 2012.

[59] Alexander Fromm, Florian Kelbert, and Alexander Pretschner. Data protection in a cloud-enabled smart grid. In *SmartGridSec*, pages 96–107, 2013.

[60] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[61] Richard Gay, Heiko Mantel, and Barbara Sprick. Service automata. In *Proceedings of the 8th International Conference on Formal Aspects of Security and Trust*, FAST'11, pages 148–163, Berlin, Heidelberg, 2012. Springer-Verlag.

[62] Gabriela Gheorghe, Paolo Mori, Bruno Crispo, and Fabio Martinelli. Enforcing ucon policies on the enterprise service bus. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II*, OTM'10, pages 876–893, Berlin, Heidelberg, 2010. Springer-Verlag.

[63] Gabriela Gheorghe, Stephan Neuhaus, and Bruno Crispo. xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In Masakatsu Nishigaki, Audun Jøsang, Yuko Murayama, and Stephen Marsh, editors, *Trust Management IV*, volume 321 of *IFIP Advances in Information and Communication Technology*, pages 63–78. Springer Berlin Heidelberg, 2010.

[64] D. Giffhorn. *Slicing of Concurrent Programs and its Application to Information Flow Control*. PhD thesis, Karlsruher Institut für Technologie, 2012.

[65] Dennis Giffhorn and Christian Hammer. Precise slicing of concurrent programs. *Automated Software Engineering*, 16(2):197–234, 2009.

[66] Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, pages 1–25, 2014.

[67] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[68] Joseph A. Goguen and José Meseguer. Unwinding and inference control. *IEEE Symposium on Security and Privacy*, 0:75, 1984.

[69] J. Graf. *Information Flow Control with SDGs — Improving Modularity, Scalability and Precision for Object Oriented Languages*. PhD thesis, KIT. Forthcoming.

[70] J. Graf, M. Hecker, and M. Mohr. Using joana for information flow control in java programs - a practical guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS 2013)*, 2013.

[71] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[72] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, October 2009.

[73] 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*, pages 373–380, Oct 2009.

[74] Klaus Havelund and Grigore Rosu. Synthesizing monitors for safety properties. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 2002.

[75] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)*, 6, Aug 2004.

[76] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proceedings of the 12th European Conference on Research in Computer Security*, ESORICS'07, pages 531–546, Berlin, Heidelberg, 2007. Springer-Verlag.

[77] M. Hilty, A. Pretschner, C. Schaefer, C. Schaefer, and T. Walter. Usage Control Requirements in Mobile and Ubiquitous Computing Applications. In *Systems and Networks Communications, 2006. ICSNC '06. International Conference on*, pages 27–27, Oct 2006.

[78] Manuel Hilty. *Specification and Enforcement in Distributed Usage Control*. PhD thesis, ETH Zürich, 2008.

[79] Manuel Hilty, David Basin, and Alexander Pretschner. On obligations. In *Proceedings of the 10th European Symposium on Research in Computer Security*, ESORICS'05, pages 98–117, Berlin, Heidelberg, 2005. Springer-Verlag.

[80] Manuel Hilty, Alexander Pretschner, and Felix Akeret. Anforderungen für verteilte nutzungskontrolle, 2005. SIEMENS AG (CH) internal report.

[81] Manuel Hilty, Alexander Pretschner, Christian Schaefer, and Thomas Walter. Enforcement for usage control — a system model and an obligation language for distributed usage control. Technical Report I-ST-020, DOCOMO Euro-Labs, 2006.

[82] Manuel Hilty, Alexander Pretschner, Thomas Walter, and Christian Schaefer. Usage Control Requirements in Mobile and Ubiquitous Computing Applications. Technical Report I-ST-015, DOCOMO Euro-Labs, 2005.

[83] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 29–41, New York, NY, USA, 2006. ACM.

[84] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[85] James W. Gray III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35, 1991.

[86] Iulia Ion, Boris Dragovic, and Bruno Crispo. Extending the java virtual machine to enforce fine-grained security policies in mobile devices. In *Proceedings of ACSAC 2007 (the 23rd Annual Computer Security Applications Conference)*, Miami Beach, Florida, December 2007.

[87] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[88] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[89] Christian Jung, Denis Feth, and Christian Seise. Context-aware policy enforcement for android. In *Proceedings of the 2013 IEEE 7th International Conference on Software Security and Reliability*, SERE '13, pages 40–49, Washington, DC, USA, 2013. IEEE Computer Society.

[90] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2011.

[91] Florian Kelbert and Alexander Pretschner. Data usage control enforcement in distributed systems. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 71–82, New York, NY, USA, 2013. ACM.

[92] Florian Kelbert and Alexander Pretschner. Decentralized distributed data usage control. In Dimitris Gritzalis, Aggelos Kiayias, and Ioannis Askoxylakis, editors, *Cryptology and Network Security*, volume 8813 of *Lecture Notes in Computer Science*, pages 353–369. Springer International Publishing, 2014.

[93] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[94] Hyung Chan Kim, Angelos D. Keromytis, Michael Covington, and Ravi Sahita. Capturing information flow with concatenated dynamic taint analysis. In *Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan*, pages 355–362, 2009.

[95] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[96] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *SIGOPS Oper. Syst. Rev.*, 41(6):321–334, October 2007.

[97] Prachi Kumari, Florian Kelbert, and Alexander Pretschner. Data protection in heterogeneous distributed systems: A smart meter example. In *Dependable Software for Critical Infrastructures*, October 2011.

[98] Prachi Kumari and Alexander Pretschner. Deriving implementation-level policies for usage control enforcement. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 83–94, New York, NY, USA, 2012. ACM.

[99] Prachi Kumari and Alexander Pretschner. Model-based usage control policy derivation. In *Proceedings of the 5th International Conference on Engineering Secure Software and Systems*, ESSoS'13, pages 58–74, Berlin, Heidelberg, 2013. Springer-Verlag.

[100] Prachi Kumari, Alexander Pretschner, Jonas Peschla, and Jens-Michael Kuhn. Distributed data usage control for web applications: A social network implementation. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pages 85–96, New York, NY, USA, 2011. ACM.

[101] Orna Kupferman and Moshe Y. Vardi. Model checking of safety properties. *Form. Methods Syst. Des.*, 19(3):291–314, October 2001.

[102] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, May 2009.

[103] Matthias Leumann. Policy evaluation and negotiation in distributed usage control. Master's thesis, Department of Informatics, ETH Zürich, 2007.

[104] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, volume 3923 of *Lecture Notes in Computer Science*, pages 47–64. Springer Berlin Heidelberg, 2006.

[105] Michael Lörscher. Usage control for a mail client. Master's thesis, Department of Informatics, Kaiserslautern University of Technology, 2012.

[106] Volkmar Lotz, Emmanuel Pigout, Peter M. Fischer, Donald Kossmann, Fabio Massacci, and Alexander Pretschner. Towards systematic achievement of compliance in service-oriented architectures: The master approach. *WIRTSCHAFTSINFORMATIK*, 50(5):383–391, 2008.

[107] Enrico Lovat, Alexander Fromm, Martin Mohr, and Alexander Pretschner. SHRIFT System-wide HybRid Information Flow Tracking. In *ICT Systems Security and Privacy Protection*, IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2015.

[108] Enrico Lovat and Florian Kelbert. Structure matters - A new approach for data flow tracking. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA, May 17-18, 2014*, pages 39–43, 2014.

[109] Enrico Lovat, Martín Ochoa, and Alexander Pretschner. Cross-layer data flow tracking (submitted). 2015.

[110] Enrico Lovat, Johan Oudinet, and Alexander Pretschner. On quantitative dynamic data flow tracking. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 211–222, New York, NY, USA, 2014. ACM.

[111] Enrico Lovat and Alexander Pretschner. Data-centric multi-layer usage control enforcement: A social network example. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 151–152, New York, NY, USA, 2011. ACM.

[112] David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, volume 5321 of *Lecture Notes in Computer Science*, pages 3–3. Springer Berlin Heidelberg, 2008.

[113] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[114] Heiko Mantel. Possibilistic definitions of security - an assembly kit. *Computer Security Foundations Workshop, IEEE*, 0:185, 2000.

[115] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. *SIGPLAN Not.*, 43(6):193–205, June 2008.

[116] Microsoft. Active Directory Rights Management Services Windows. `https://technet.microsoft.com/en-us/library/dn339006(v=ws.10).aspx`(last access: 11.05.2015), 2015.

[117] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 305–315, New York, NY, USA, 2010. ACM.

[118] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[119] Jonathan K. Millen. Covert channel capacity. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 27-29, 1987*, pages 60–66, 1987.

[120] Cornelius Moucha, Enrico Lovat, and Alexander Pretschner. A hypervisor-based bus system for usage control. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES)*, pages 254–259, August 2011.

[121] Cornelius Moucha, Enrico Lovat, and Alexander Pretschner. A virtualized usage control bus system. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 2(4):84–101, 2011.

[122] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. Layering in provenance systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[123] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[124] Srijith K. Nair, Andrew S. Tanenbaum, Gabriela Gheorghe, and Bruno Crispo. Enforcing drm policies across applications. In *Proceedings of the 8th ACM workshop on Digital rights management*, DRM '08, pages 87–94, New York, NY, USA, 2008. ACM.

[125] Mangala Gowri Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, November 2006.

[126] Ricardo Neisse, Alexander Pretschner, and Valentina Di Giacomo. A trustworthy usage control enforcement framework. *2012 Seventh International Conference on Availability, Reliability and Security*, 0:230–235, 2011.

[127] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[128] Open Mobile Alliance. DRM Rights Expression Language V2.1, 2008. `http://www.openmobilealliance.org/Technical/release_program/drm_v2_1.aspx`(last access: 11.05.2015).

[129] Jaehong Park and Ravi Sandhu. The UCONABC Usage Control Model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004.

[130] Miodrag Petkovic, M. Popovic, Ilija Basicevic, and Djordje Saric. A host based method for data leak protection by tracking sensitive data flow. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pages 267–274, April 2012.

[131] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[132] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for usage control. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 240–244, New York, NY, USA, 2008. ACM.

[133] A. Pretschner, M. Hilty, F. Schutz, C. Schaefer, and T. Walter. Usage control enforcement: Present and future. *Security Privacy, IEEE*, 6(4):44–53, July 2008.

[134] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy evolution in distributed usage control. *Electron. Notes Theor. Comput. Sci.*, 244:109–123, August 2009.

[135] Alexander Pretschner. An Overview of Distributed Usage Control. In *Knowledge Engineering: Principles and Techniques Conference*, pages 25–33, 2009.

[136] Alexander Pretschner, Matthias Büchler, Matus Harvan, Christian Schaefer, and Thomas Walter. Usage control enforcement with data flow tracking for x11. In *Proceedings of the 5th International Workshop on Security and Trust Management*, pages 124–137, 2009.

[137] Alexander Pretschner, Florian Kelbert, Prachi Kumari, Enrico Lovat, and Tobias W uechner. *Distributed Data Usage Control*. Springer, 2015.

[138] Alexander Pretschner, Enrico Lovat, and Matthias Büchler. Representation-independent data usage control. In *Proceedings of the 6th International Conference, and 4th International Conference on Data Privacy Management and Autonomous Spontaneus Security*, DPM'11, pages 122–140, Berlin, Heidelberg, 2012. Springer-Verlag.

[139] Alexander Pretschner, Judith Rüesch, Christian Schaefer, and Thomas Walter. Formal analyses of usage control policies. In *ARES*, pages 98–105, 2009. Policy descriptions for experiments available at `http://www22.in.tum.de/fileadmin/papers/ares09-experiments.pdf`.

[140] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. *Proceedings of the Network and Distributed System Security Symposium (NDSS 2014)*, 2014.

[141] Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. DroidForce: Enforcing Complex, Data-Centric, System-Wide Policies in Android. In *Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES)*. IEEE, September 2014.

[142] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. *ACM SIGSOFT Software Engineering Notes*, 19(5):11–20, December 1994.

[143] Erik Rissanen. Extensible access control markup language v3.0. `http://docs.oasis-open.org/xacml/3.0/`(last access: 11.05.2015), 2010.

[144] Bruno P. S. Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo. Hybrid static-runtime information flow and declassification enforcement. *Trans. Info. For. Sec.*, 8(8):1294–1305, August 2013.

[145] John Rushby. Noninterference, transitivity and channel-control security policies, 1992.

[146] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

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

[148] R.S. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40–48, Sept 1994.

[149] Shakti Saxena. Data usage control in office application. Master's thesis, Department of Informatics, Technische Universität München, 2014.

[150] O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 164–174, New York, NY, USA, 1988. ACM.

[151] Asia Slowinska and Herbert Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 61–74, New York, NY, USA, 2009. ACM.

[152] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. *ACM SIGPLAN Notices - ASPLOS*, 39(11):85–96, October 2004.

[153] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[154] Kevin Twidle, Emil Lupu, Naranker Dulay, and Morris Sloman. Ponder2 - a policy environment for autonomous pervasive systems. In *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*, POLICY '08, pages 245–246, Washington, DC, USA, 2008. IEEE Computer Society.

[155] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, Blome J. A., G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, 2004.

[156] Dries Vanoverberghe and Frank Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In Gilles Barthe and FrankS. de Boer, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 240–258. Springer Berlin Heidelberg, 2008.

[157] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.

[158] Dennis M. Volpano. Safety versus secrecy. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 303–311, London, UK, UK, 1999. Springer-Verlag.

[159] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '97, pages 607–621, London, UK, UK, 1997. Springer-Verlag.

[160] W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification, 2005. `http://www.w3.org/TR/2005/WD-P3P11-20050104/`(last access: 11.05.2015).

[161] Ting Wang, Mudhakar Srivatsa, Dakshi Agrawal, and Ling Liu. Modeling data flow in socio-information networks: A risk estimation approach. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 113–122, New York, NY, USA, 2011. ACM.

[162] Daniel Wasserrab and Denis Lohner. Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In *6th International Verification Workshop - VERIFY-2010*, 2010.

[163] Patrick Wenz. Data Usage Control for ChromiumOS. Master's thesis, Department of Informatics, Karlsruhe Institute of Technology, 2012.

[164] Tobias Wüchner and Alexander Pretschner. Data loss prevention based on data-driven usage control. In *23rd IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 151–160, Nov 2012.

[165] Tobias Wüechner. Implementation of usage control for the windows api. Master's thesis, Department of Informatics, Kaiserslautern University of Technology, 2011.

[166] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society.

[167] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[168] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

[169] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[170] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[171] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 369–382, Berkeley, CA, USA, 2013. USENIX Association.

[172] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 63–74, New York, NY, USA, 2010. ACM.

[173] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A logical specification for usage control. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 1–10, New York, NY, USA, 2004. ACM.

[174] Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A logical specification for usage control. In *Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 1–10, New York, NY, USA, 2004. ACM.

[175] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *SIGOPS Operating System Review*, 45(1):142–154, February 2011.