Sound and Precise Cross-Layer Data Flow Tracking

Enrico Lovat, Martin Ochoa, Alexander Pretschner

TUM-I1629

Sound and Precise Cross-Layer Data Flow Tracking

Enrico Lovat[1], Martín Ochoa[2] and Alexander Pretschner[1]

[1]Technische Universität München, Germany
[2]Singapore University of Technology and Design

Abstract. We present a general model that connects runtime monitors for data flow tracking at different abstraction layers (e.g. a browser, a mail client, an operating system) and show that this model is sound with respect to a formal notion of explicit information flow. This is useful because, although the semantics of higher-level events at a single layer can be exploited to increase the precision of the analysis, other abstraction levels need to be considered as well in order to obtain system-wide guarantees. For instance, to soundly reason about the flow of a picture from the network through a browser into a cache file or a window on the screen, analysis results from multiple layers need to be combined.

1 Introduction

Research in data flow tracking [4,21] tackles the problem of monitoring flows of data from sources (i.e. input parameters to methods, sockets, files) to sinks (i.e. outputs to sockets, files). Data flow analysis systems can answer the question if data has (potentially) flowed, or will (potentially) flow, from a source to a sink.

Dynamic approaches for data-flow tracking implement reference monitors at various levels of abstraction: binary code, Java bytecode, operating systems, and dedicated applications. Dynamic analyses can exploit layer-specific semantic information and be precise in the presence of reflection or call-backs. They cannot, by definition, detect flows that are a consequence of non-executed branches and they do impose a non-negligible runtime overhead. In the absence of OS-layer monitoring and if monitoring is not done at the binary level, dynamic analysis results are confined to the considered layer. In this paper, we elaborate on the idea of using multiple monitors at different layers, with the goal of improving the precision of the single layers by exploiting known relations between them.

As an example, consider Fig. 1 where an application loads two files from the OS and then saves one of them with a different name. Data d, contained in the first file *file* f, enters the application via container *src1* (1), is propagated through the application internals (2) and finally leaves the application (3). If dynamic monitoring was performed solely at the OS layer, the analysis would report data e to have flowed to *file* i as well— which is sound but over-approximating. If monitoring was solely performed at the application level, data flows at the OS layer could not be observed, e.g., the flow of data d from *file* i to *file* h.

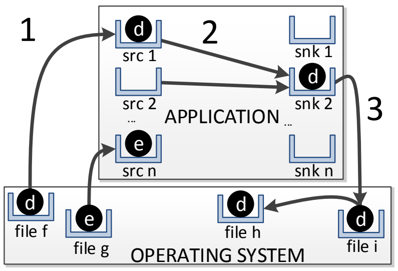

Fig. 1. Intra-app data flow example.

Using monitors at multiple layers, two in this example, allows us to increase precision of single-layer analyses, to detect and control system-wide (rather than program-wide) data flows, and to exploit semantic information at various layers.

It has been shown [7,25,26] that dynamic analyses can be performed at the level of every piece of binary code that is executed. This obviates the need for multiple layers. However, in addition to overhead issues of purely dynamic solutions, at low levels it is also hard to detect and model semantics of high-level events (like "take a screenshot") or objects (like "a mail"). While implementations of such system-wide information-flow trackers exist [7,25,26], we are not aware of any formal description of the guarantees that can be provided. Using monitors at levels different from the binary code, we decrease the number of events that need to be intercepted and exploit semantic information of the single levels.

Explicit vs. Implicit flows: From an information flow perspective it is important to consider both explicit and implicit flows (as in non-interference [9]). This work, however, focuses on *explicit* data flows only, with the well-known limitations that this choice entails. The reason is twofold: first, we want to take advantage of the *semantic* information gained by monitoring high-level events (e.g. "play" or "forward") and those events typically involve only explicit data flows (such as copying data from a file to another). Second, many implicit flows are harmless in practice (i.e. the famous password-checking example [20] among others), such that by enforcing strict non-interference one is likely to severely hamper the intended functionality of a system. Enhancing our cross-layer analysis to include some implicit information flows while preserving functionality requires considering non-trivial declassification policies and is left for future work.

The **problem** that we tackle is the following. We assume that there are dynamic analyses at two or more levels, all of them different from that of CPU-level instructions, including operating system, application level, database, programming language and window manager. How can we connect the analysis results of the different layers, and what guarantees can we give?

Our **contributions** are **(a)** a formal definition of soundness of data flow tracking for single levels different from that of the CPU; **(b)** a formal definition of soundness of data flow tracking when multiple layers are combined; **(c)** a generic schema to compose data flow analyses at various levels and that thus enables us to detect system-wide data flows; and **(d)** a proof that this generic schema returns sound data flow results, provided that the single layers are correct and given some partial information about shared resources at both layers.

We do not discuss implementations of this model here, but we point to examples in § 7, where some strategies for cross-layer composition described here have been applied without a proper formal justification and with implicit assumptions that we now make explicit. See [15] for more details.

The rest of the paper is organized as follows: § 2 introduces fundamental concepts and sets the notation for the rest of the paper. § 3 discusses the security guarantees for single layers. § 4 defines layer composition and extends the soundness notion to composed systems. § 5 presents the main result of the paper, i.e. an algorithm for soundly composing monitor results at different abstraction layers. In § 6 we review related work and we conclude in § 7.

2 Background and Roadmap

We consider flows in systems described as tuples $(\mathcal{E}, \mathcal{D}, \mathcal{C}, \Sigma, \sigma_i, \mathcal{R})$ for system events $\mathcal{E}$, data items $\mathcal{D}$ (e.g., "a picture"), and containers $\mathcal{C}$ (these are representations: a pixmap, a file, a memory region, a set of network packets, and so on). For the time being, these systems can be understood as single layers. In the following we assume that the alphabet of system events $\mathcal{E}$, the set of data items $\mathcal{D}$ and the set of containers $\mathcal{C}$ can be of arbitrary but finite size.

States Σ are defined by a *storage function* of type $\mathcal{C} \to \mathbb{P}(\mathcal{D})$ that describes which set of data is potentially stored in which container. Data items will be often referred to as *labels* in the following.

The transition function $\mathcal{R} : \Sigma \times \mathcal{E} \to \Sigma$ is the core of the data-flow tracking model, encoding how the execution of events affects the dissemination of data in the system (and therefore also referred to as a *monitor* in the following). At runtime, events are intercepted and the data state is updated according to $\mathcal{R}$. $\mathcal{R}$ applied to a sequence of events is the recursive application to each event in the sequence (i.e. $\mathcal{R}(\sigma, \langle e_1, \langle ... \rangle\rangle) = \mathcal{R}(\mathcal{R}(\sigma, e_1), \langle ... \rangle)$.

Abstraction layers: We will show desirable properties by relating the model for one layer A to a very low level model $\perp$ with intuitive completeness and correctness properties. Layer A can be an operating system, a data base, a windowing system, an application, etc. $\perp$ is the level of the CPU and volatile as well as persistent memory cells, and represents the *real* execution of the system. Let $\mathcal{V}$ be the set of all total functions of type $\mathcal{C}_\perp \to \mathbb{N}$ that map containers to actual *values* stored in memory. We provide level $\perp$ with a function $v \in \mathcal{V}$, that indicates the current state of memory, and a trace execution semantics $\mathsf{eval} : \mathcal{V} \times seq(\mathcal{E}_\perp) \to \mathcal{V}$ that describes the state after executing a trace, such that the system at $\perp$ is given by $(\mathcal{E}_\perp, \mathcal{D}_\perp, \mathcal{C}_\perp, \Sigma_\perp, \sigma_i, \mathcal{R}_\perp, \mathcal{V}, \mathsf{eval})$.

A in contrast is some distinct higher layer. Set $\mathcal{L}$ denotes the set of all these high levels, while $\mathcal{L}^\perp = \mathcal{L} \cup \{\perp\}$. Data $\mathcal{D}$ is layer-independent. For $\dagger \in \mathcal{L}$, $\mathcal{C}_\dagger$ denotes the set of representations of some data item at layer $\dagger$. To relate two layers, we assume pairs of functions γ and α that relate events and containers as follows. The idea is that an A-level container corresponds to a set of $\perp$-level containers (volatile and persistent memory cells) and an A-level action to a sequence of *CPU*-level instructions (machine instructions such as `MOV, BNE, ADD, LEQ`). For a layer $\dagger \in \mathcal{L}^\perp$, each state $\sigma_\dagger \in \Sigma_\dagger$ is defined by the respective storage function. Additionally, for $\perp$, $v \in \mathcal{V}$ encodes the memory state.

Relating events and states: In the following we will introduce abstraction and concretization functions that relate events and states at a higher abstraction level with $\perp$. This notation will allows us to define formal soundness properties. Note that our goal is to reason about what happened at the $\perp$ level, *without monitoring it*, by assuming partial information on the abstraction/concretization functions. This will be made operationally in practice by *oracles*, as we will discuss in § 5. For the moment, concretization and abstraction functions are *ideal*: they relate to "what has really happened" in a monitored system, and where for instance scheduling of concurrent processes has already been fixed.

Traces are sequences of events that reflect the execution of some functionality at a specified layer. *Traces* are sequences of events that reflect the execution of some functionality at a specified layer. A trace $t \in seq(\mathcal{E}_\dagger)$ at layer $\dagger \in \mathcal{L}$ is thus the sequence of events generated by the system during a specific run (e.g "*print x*" or "`mov AX,BX`"). At the $\perp$-level, traces are mere sequences of CPU instructions. We assume events to be unique to exactly one level 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. For simplicity's sake, we assume that we can bijectively map an abstract sequence of events to a concrete sequence of $\perp$-events. This embodies the fundamental assumption of a *single-core system*: all traces can be uniquely sequentialized. While a single-core can simulate parallelism alternating execution of different tasks, we assume that events at high levels happen *sequentially* (e.g. first save $file_1$ and then load $file_2$). This allows us to use a simpler notation in these introductory sections. In § 5 we relax this assumption, because serializable traces can capture also concurrent executions.

Moreover, we deliberately *discard events at $\perp$ that do not correspond to an event at a higher layer*, e.g. those generated by an application for which there is no explicit monitor. The implication is that our approach can only be sound w.r.t. those CPU-level instructions for which a monitor at some level exists.

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

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

such that

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

Additionally, $\forall \mathcal{C} \subseteq \mathcal{C}_\dagger : \gamma_\dagger(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \gamma_\dagger(c)$ and $\forall \mathcal{C} \subseteq \mathcal{C}_\perp : \alpha_\dagger(\mathcal{C}) = \bigcup_{c \in \mathcal{C}} \alpha_\dagger(c)$. For each layer $\dagger \in \mathcal{L}$ we assume the existence of a special container $c_\dagger^U$ that represents the abstraction of all those $\perp$-level containers not observable at the $\dagger$-level $(\forall c_\perp \in \mathcal{C}_\perp : (\forall c_\dagger \in \mathcal{C}_\dagger \setminus \{c_\dagger^U\} : \alpha_\dagger(c_\perp) \neq c_\dagger) \implies (\alpha_\dagger(c_\perp) = c_\dagger^U))$. By definition $\sigma(c_\dagger^U) = \mathcal{D}$ for any state σ.

State and trace union: Given two states σ_1 and σ_2 at the same abstraction level, let $\sigma_1 \bowtie_\sigma \sigma_2 = \{(c, D) \mid D = \sigma_1(c) \cup \sigma_2(c)\}$. Recall that events at any level are assumed to be unique and to contain an implicit timestamp, yielding a natural order on a trace's events. We denote by $t_1 \bowtie_t t_2$ the time-ordered trace consisting of unique elements of t_1 and t_2.

Roadmap. In the following we give a high-level account on the strategy used in the rest of the paper to justify the proposed cross-layer algorithm.

(1) We relate the notion of taint propagation at the lowest abstraction layer ($\perp$) with that of *weak secrecy*. **(2)** We define a notion of *soundness* for a single layer A with respect to $\perp$. The intuition behind this definition is that the taint propagation in A (specified by $\mathcal{R}_A$) must be coherent with respect to taint propagation happening at $\perp$. This definition offers a semantic characterization for single monitors at any level. **(3)** We then define composed states $\sigma_{A\otimes B}$ as pairs of states (σ_A, σ_B) at different layers and give a notion of sound composed monitor. **(4)** We construct and prove the soundness of a composed monitor $\hat{\mathcal{R}}_{A\otimes B}$ that relies on the soundness of monitors $\mathcal{R}_A$ and $\mathcal{R}_B$ at the single layers and on partial information on γ. **(5)** We show an example where additional information about A and B can lead to a more precise cross-layer tracking and **(6)** we use it to motivate the usefulness of further *oracles* that encode partial information about γ and α. **(7)** We construct a composed monitor $\dot{\mathcal{R}}_{A\otimes B}$ that relies on the soundness of single layer monitors $\mathcal{R}_A$ and $\mathcal{R}_B$ and on the information from the oracles and show its soundness.

Thanks to the above construction, we can connect existing data-flow tracking analyses for different layers of abstraction (e.g. [24,13,16]) to capture data flows across layers, and show overall soundness, or weak secrecy, respectively.

3 Security guarantees at single layers

In the following, we define the notion of information flow which will be the fundamental security property guaranteed by our framework. We use this notion to show soundness of the propagating data flow monitors at various layers.

3.1 Step 1: Security property at the $\perp$ 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 [9], which relates inputs and outputs in terms of pairs of executions (or state of variables before and after executing a program [23]).

Definition 1 (Non-interference). *Let $\mathcal{C}_H^i, \mathcal{C}_H^o \subseteq \mathcal{C}_\perp$ be sets of containers at $\perp$ and $\mathcal{C}_L^i, \mathcal{C}_L^o$ their complements. A trace $t_\perp \in seq(\mathcal{E}_\perp)$ respects* non-interference *w.r.t. this partition of the containers if*

$$\forall v, v' \in \mathcal{V} : \bigwedge_{c \in \mathcal{C}_L^i} v(c) = v'(c) \Longrightarrow \bigwedge_{c \in \mathcal{C}_L^o} \mathsf{eval}(v, t_\perp)(c) = \mathsf{eval}(v', t_\perp)(c)$$

low) containers, are independent from its complement (the *high* containers) after execution of a trace at $\perp$. This represents the notion of *absence of flows* from high to low containers. As discussed in the introduction, in this work we focus on explicit information flows. A relaxed notion of non-interference which captures such flows is *weak-secrecy* [22].

To formally define this property in our context, consider a trace $t_\perp \in seq(\mathcal{E}_\perp)$. We say that its *branch-free* version $bf(t_\perp)$ consists of the same assembly-level instructions in the same order, except for branch statements such as `BNE` (branch-non-equal), which are removed from the observed trace. Of course there are no *branches* in one actually executed trace. There are, however, conditional jumps like *branch-not-equal* that may lead to *implicit, or control-flow-based* information flows. In order to cater to explicit flows only, these instructions are ignored, i.e. removed. The resulting trace then corresponds to one *path* through the CFG of the original program where all conditional nodes are replaced by empty statements. By doing so, our notion of security becomes the verification of non-interference on a sequence of explicit data flows only.

Definition 2 (Weak secrecy). *Let $\mathcal{C}_H^i, \mathcal{C}_H^o \subseteq \mathcal{C}_\perp$ be sets of containers at $\perp$. A trace $t_\perp \in seq(\mathcal{E}_\perp)$ respects* weak-secrecy *w.r.t. $\mathcal{C}_H^i, \mathcal{C}_H^o$ if its branch-free version $bf(t)$ respects non-interference w.r.t. $\mathcal{C}_H^i, \mathcal{C}_H^o$.*

Note that non-interference in general does not imply weak secrecy. Moreover, our construction is not intended to guarantee non-interference: we need it to define weak secrecy only. A monitor $\mathcal{R}_\perp$ propagates labels (i.e. data items) in-between containers as the consequence of the execution of a trace.

Definition 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 $t_\perp \in seq(\mathcal{E}_\perp)$ 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 following, $\mathcal{R}_\perp^\#$ indicates the (virtual) most precise sound monitor at level $\perp$, i.e. for all $d \in \mathcal{D}$ and $t_\perp \in seq(\mathcal{E}_\perp)$, the output partition $\mathcal{C}_H^o$ induced by any sound monitor includes that induced by $\mathcal{R}_\perp^\#$.

Sources and destinations From the point of view of $\mathcal{R}_\perp^\#$, events move data from a container to another: an instruction typically reads from a certain memory region and writes to another. We say that for any given event e and a transition function $\mathcal{R}_\perp^\#$, the functions $\mathbb{S}_{\mathcal{R}_\perp^\#} : \mathcal{E}_\perp \to 2^{\mathcal{C}_\perp}$ and $\mathbb{D}_{\mathcal{R}_\perp^\#} : \mathcal{E}_\perp \to 2^{\mathcal{C}_\perp}$ denote, respectively, the set of source and the set of destination containers of the events. We assume the two functions to be given as an oracle of the event, such that for all $\perp$-containers c, states σ and data items d,

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

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

Note that there could be coarse partitions that fulfill this property. In the following we assume that the oracle provides the most precise ones in the sense that e respects weak secrecy w.r.t. $\mathbb{S}_{\mathcal{R}_\perp^\#}(\mathrm{e}), \mathcal{C}_\perp \backslash \mathbb{D}_{\mathcal{R}_\perp^\#}(e)$, and w.r.t $\mathcal{C}_\perp \backslash \mathbb{S}_{\mathcal{R}_\perp^\#}(e), \mathbb{D}_{\mathcal{R}_\perp^\#}(e)$. In other words, there is non-interference between the partitions induced by sources and destinations and their dual complement. Intuitively, this ensures that all relevant sources and all relevant destinations are captured and no more.

We overload the notation of $\mathbb{S}$ and $\mathbb{D}$ for traces of events $t \in seq(\mathcal{E}_\perp)$ as $\mathbb{S}_{\mathcal{R}_\perp}(t) = \bigcup_{e \in t} \mathbb{S}_{\mathcal{R}_\perp^\#}(e)$ and $\mathbb{D}_{\mathcal{R}_\perp^\#}(t) = \bigcup_{e \in t} \mathbb{D}_{\mathcal{R}_\perp^\#}(e)$. A similar overloading applies to sets of events. We also extend the notation for events and monitors at higher layers of abstraction, $\mathbb{S}_{\mathcal{R}_\dagger}$ and $\mathbb{D}_{\mathcal{R}_\dagger}$ for $\dagger \in \mathcal{L}$, such that the same relation between $\mathcal{R}_\dagger$, containers and data holds at level $\dagger$.

3.2 Step 2: Soundness at a single layer

An A-level state of the system, σ_A, is sound if, for every container c_A, the set of data stored in c_A is a superset 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. In the following, we assume a fixed pair of γ_A/α_A w.r.t. which soundness is defined.

Definition 4. *A state σ_A is* sound w.r.t. $\sigma_\perp$, *written $\sigma_\perp \vdash \sigma_A$, iff*

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

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. $\perp$ (i.e., it respects weak-secrecy) if $\mathcal{R}_A$ preserves the soundness of the state (w.r.t. the canonical $\mathcal{R}_\perp^\#$ of Definition 3).

Definition 5 (Soundness of single layer monitor). *A monitor $\mathcal{R}_A$ at a level A is sound w.r.t. $\perp$, written $\mathcal{R}_\perp^\# \vdash \mathcal{R}_A$, if given an initial state $\sigma_\perp^i \vdash \sigma_A^i$, modeling any trace of events $t_A \in seq(\mathcal{E}_A)$ 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,* $\forall t_A \in seq(\mathcal{E}_A), \sigma_A^i \in \Sigma_A, \sigma_\perp^i \in \Sigma_\perp : \mathcal{R}_\perp^\# \vdash \mathcal{R}_A \iff \sigma_\perp^i \vdash \sigma_A^i \wedge \mathcal{R}_\perp^\#(\sigma_\perp^i, \gamma_A(t_A)) \vdash \mathcal{R}_A(\sigma_A^i, t_A)$.

As direct corollary, $\mathbb{S}_{\mathcal{R}_\perp^\#}(\gamma_A(e)) \subseteq \gamma_A(\mathbb{S}_{\mathcal{R}_A}(e))$ and $\mathbb{D}_{\mathcal{R}_\perp^\#}(\gamma_A(e)) \subseteq \gamma_A(\mathbb{D}_{\mathcal{R}_A}(e))$.

4 Guarantees for Multiple Layers

A monitoring infrastructure is unsound if there exists a container at a higher abstraction layer that ignores the presence of data in its concretization. Unless one performs tracking at the level of single machine instructions, this situation is likely, and we cannot expect to achieve system-wide soundness in practice. What we can do, however, is to show *cross-layer soundness*: under the strong assumption of having sound models of two (or n) layers, we can show that if data moves exclusively within or in-between these layers, and we have information about their shared resources, our cross-layer model captures all cross-layer flows.

4.1 Step 3: Layer composition

We proceed to define a notion of layer composition and discuss possible ways in which events observable at one layer may interfere with another layer. We then show a first overly-conservative way to model composition and prove its soundness. In the following, we focus on a system composed by two layers only; n-layered systems can be modeled by applying the same concepts recursively to each further layer. Without loss of generality we 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, our goal is to define a sound model for the system composed by A and B, denoted $A \otimes B$.

We begin by defining the composed system 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 $\mathcal{T}_{A\otimes B} \subseteq seq(\mathcal{E}_A) \times seq(\mathcal{E}_B)$ the set of event traces, given by pairs of traces in A and B. We denote the composed state $\sigma_{A\otimes B} \in \Sigma_{A\otimes B} \subseteq \Sigma_A \times \Sigma_B$ as a pair of states in layers A and B respectively. For this notion of states, we derive an ideal (w.r.t $\perp$) composed monitor given by concretization and abstraction functions.

When talking about the state of the system or about traces in a multilayered system $A \otimes B$, we use the notation $|_{\dagger}$ to denote the projection to layer $\dagger$.

Mathematically speaking, it is simple to compose two monitors as follows.

Definition 6 (Ideal composed monitor). *Let t_A and t_B be traces at layers A and B, respectively, and σ_A^i and σ_B^i initial sound states. Let $\sigma_{A\otimes B}^{\perp} = \gamma_A(\sigma_A^i) \bowtie_{\sigma} \gamma_B(\sigma_B^i)$, $t_{A\otimes B}^{\perp} = \gamma_A(t_A) \bowtie_t \gamma_B(t_B)$ and $\sigma'^{\perp}_{A\otimes B} = \mathcal{R}_{\perp}^{\#}(\sigma_{A\otimes B}^{\perp}, t_{A\otimes B}^{\perp})$. 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:*

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

Practically speaking, we usually do 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. However, as we did for the single layers, we can characterize sound approximations of composed monitors.

Definition 7 (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 $\sigma' = \mathcal{R}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))$ the projections to A-level containers $\sigma'|_A$ and B-level containers $\sigma'|_B$ are sound w.r.t. $\mathcal{R}_{\perp}^{\#}(\gamma_A(\sigma_A) \bowtie_{\sigma} \gamma_B(\sigma_B), \gamma_A(t_A) \bowtie_t \gamma_B(t_B))$.*

4.2 Step 4: Sound monitor based on state relation

Let two containers at different layers c_A and c_B be *related*, written $c_A \sim c_B$, if their $\perp$-concretizations overlap ($\gamma_A(c_A) \cap \gamma_B(c_B) \neq \emptyset$). Without any additional information about related containers in 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 some data d may be transferred to a container c_B by some event $e_B \in \mathcal{E}_B$, and if $c_A \sim c_B$, d would also be stored in $\gamma_A(c_A)$ because of the non-empty intersection of the concretizations. Unless d is stored in c_A, this is a violation of the soundness of $A \otimes B$ (cf. Definition 4).

However, assuming information about related containers to be known (see § 5.3), and leveraging source and sink operators, it is easy to build a sound monitor $\hat{\mathcal{R}}_{A\otimes B}$ that conservatively approximates all the data-flows induced by a trace of events by propagating the data from every source of the trace to any destination of the trace and to any container related to the destinations.

4.3 Formalization and Soundness proof of $\hat{\mathcal{R}}_{A\otimes B}$

Leveraging the source and destination sets, a sound monitor for the composed system $\hat{\mathcal{R}}_{A\otimes B} : \Sigma_{A\otimes B} \times \mathcal{T}_{A\otimes B} \rightarrow \Sigma_{A\otimes B}$ can be defined as follows:

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

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 1 (Soundness). *If $\sigma_\perp \vdash \sigma_A$, $\sigma_\perp \vdash \sigma_B$, $\mathcal{R}^\#_\perp \vdash \mathcal{R}_A$ and $\mathcal{R}^\#_\perp \vdash \mathcal{R}_B$, then $\hat{\mathcal{R}}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))$ is sound, i.e. $\mathcal{R}^\#_{A\otimes B} \vdash \hat{\mathcal{R}}_{A\otimes B}$.*

The intuition behind the proof is that there cannot exist a container c at A (resp. B) such that: (1) its concretizations at $\perp$ contain new labels d with respect to the initial state, (2) c is not in $\mathbb{D}_{A\otimes B}$ and (3) there is no $c \sim \tilde{c}$ with $\tilde{c} \in \mathbb{D}_{A\otimes B}$, i.e. $\hat{\mathcal{R}}_{A\otimes B}$ can only reach sound states.

Proof. Let $\sigma'_A = \hat{\mathcal{R}}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))|_A$, $\sigma^i_\perp$ the initial state at bottom, such that $\sigma^i_\perp \vdash \sigma_A$, and $t_{A\otimes B} = \gamma_A(t_a) \bowtie \gamma_B(t_b)$. Assume that there exists data d and containter c_A such that $d \notin \sigma'_A(c_A)$ but $\exists\, c_\perp \in \gamma_A(\sigma'_A(c_A)) : d \in c_\perp$. By definition of $\mathbb{S}_{\mathcal{R}^\#_\perp}$ then either (1) $d \in \gamma_A(\sigma_A)(c_\perp)$ or (2) $\exists\, c'_\perp \in \mathbb{S}_{\mathcal{R}^\#_\perp}(t_{A\otimes B}) : d \in c'_\perp$ and $c_\perp \in \mathbb{D}_{\mathcal{R}^\#_\perp}(t_{A\otimes B})$. Since $\sigma^i_\perp \vdash \sigma_A$ and the composed monitor only adds data to containers, then (2) must hold. By soundess 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 $c_\perp \in \mathbb{D}_{\mathcal{R}^\#_\perp}(t_{A\otimes B}) \subseteq \gamma_A(\mathbb{D}_{\mathcal{R}_A}(t_a)) \cup \gamma_B(\mathbb{D}_{\mathcal{R}_B}(t_b))$, and thus $\exists\, c_{A\otimes B} \in \mathbb{S}_{A\otimes B} : d \in c_{A\otimes B}$ and $\exists\, c'_{A\otimes B} \in \mathbb{D}_{A\otimes B} : c_\perp \in \gamma_\dagger(c'_{A\otimes B})$. 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. $\square$

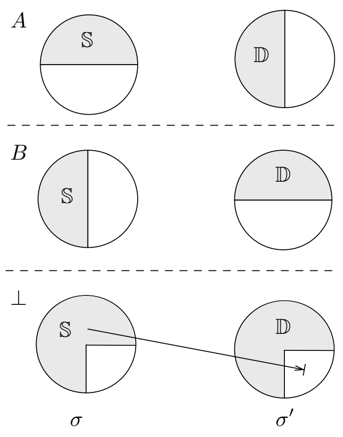

Fig. 2. Composition of potential flows between layers for a trace (t_A, t_B). Circles represent the whole memory. Partitions are induced by sources and destinations of the trace at the respective layer. Note the resulting non-interference property at $\perp$ (arrow).

As depicted in Fig. 2, there cannot be flows to memory regions outside the destination set.

$\hat{\mathcal{R}}_{A\otimes B}$ is a sound but very conservative monitor (any data read by any event in the trace is appended to the destination of any event). In the next section we see how to leverage additional information about the relation between A and B to obtain a more precise monitor for the combined system.

5 Cross-layer Models

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 such partial information in form of *oracles* and to formalize a refined data flow tracking model that, leveraging these oracles, provides more precise results. The key idea is that if more information about the relation between A and B is available, a more precise sound model can be constructed.

After extending our notation to capture the duration of events, we illustrate an example of using additional information to improve tracking precision; afterward, we abstractly define *properties* for the oracles (operationally, the oracles are implementation-specific and have to be instantiated by experts); and finally, we show an algorithm that, given two instances of the model and of the oracles, soundly approximates their composition.

Events in $A \otimes B$: In the single layers, events are assumed to be instantaneous. However, in a multi-layer context the duration of an event at one layer may span over several timesteps at the other layer. For instance, an event like `LOAD()` may be considered atomic at an application layer, although it corresponds to many system call events at the operating system layer. For this reason, it is useful to distinguish between the moment in time when an event e begins and the moment when it ends when reasoning about multiple layers.

Without loss of generality, the following assumes that every monitor for a layer $\dagger$ is defined over events in $\mathcal{E}_\dagger^- \subseteq \mathcal{E}_\dagger \times \{S, E\}$, where the suffixes S and E for an event e indicate, respectively, the beginning and the end of the execution of e. $\mathcal{R}_\dagger^-$ denotes a monitor for such traces. While for simplicity's sake, we assume high level events to happen sequentially, serialized traces could also capture interleaving of events, e.g. $\langle e_S, e'_S, e_E, e'_E \rangle$. Note that, at level $\perp$, events are still serialized (single-core assumption, § 2). To simplify the notation, whenever a trace contains a certain event e_S directly followed by e_E, we write both events as e, e.g. instead of $\langle \texttt{LOAD}_s\texttt{()}, \texttt{READ}_s\texttt{()}, \texttt{READ}_e\texttt{()}, \texttt{LOAD}_e\texttt{()} \rangle$ we write $\langle \texttt{LOAD}_s\texttt{()}, \texttt{READ()}, \texttt{LOAD}_e\texttt{()} \rangle$.

5.1 Serialized events

Let $t^S(e) : \mathcal{E} \to \mathbb{N}$ and $t^E(e) : \mathcal{E} \to \mathbb{N}$ be two functions that return, respectively, the time at which a certain event e starts and ends. In the context of multiple layers, we assume that for any event $e_\dagger \in \mathcal{E}_\dagger$ it holds that $e_\dagger$ terminates only after starting ($t^S(e_\dagger) < t^E(e_\dagger)$) and that for every event e observed, the single layer monitors report an event e^S at time $t^S(e)$ to notify the beginning of e and an event e^E at time $t^E(e)$ to notify its end. In concrete implementations it is usually possible to observe or approximate these two aspects of any event.

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

Lemma 1. *For each monitor $\mathcal{R}_\dagger$ ($\dagger \in \mathcal{L}$), there always exists a monitor $\mathcal{R}_\dagger^- : \Sigma_\dagger \times \mathcal{E}_\dagger^- \to \Sigma_\dagger$ such that $\forall \sigma_\dagger \in \Sigma_\dagger, \forall t_\dagger \in \mathit{seq}(\mathcal{E}_\dagger) : \mathcal{R}_\dagger(\sigma, t_\dagger) = \mathcal{R}_\dagger^-(\sigma, \mathit{ser}(t_\dagger))$.*

Proof. Given $\mathcal{R}_\dagger$, the monitor $\mathcal{R}_\dagger^-$, defined as $\mathcal{R}_\dagger^-(\sigma, (e_\dagger, i)) = \sigma$ if $i = S$ and $\mathcal{R}_\dagger^-(\sigma, (e_\dagger, i)) = \mathcal{R}_\dagger(\sigma, e_\dagger)$ if $i = E$, respects the property. □

It is hence safe to assume, without loss of generality, that every monitor for a layer $\dagger$ is defined over events in $\mathcal{E}_\dagger^-$. We denote such a monitor $\mathcal{R}_\dagger^-$.

Definition 8 (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 $\mathcal{E}_{A \otimes B} = \mathcal{E}_A \cup \mathcal{E}_B$ and $\mathcal{E}_{A \otimes B}^- = \mathcal{E}_{A \otimes B} \times \{S, E\}$. If a trace $t = (t_A, t_B) \in \mathit{seq}(\mathcal{E}_A) \times \mathit{seq}(\mathcal{E}_B)$ is serializable, then it is possible to construct a trace $t^- \in \mathit{seq}(\mathcal{E}_{A \otimes B}^-)$ 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 $\mathit{ser}(t_A) \bowtie_t \mathit{ser}(t_B)$ sorted by timestamp. The monitor for the composed system $\dot{\mathcal{R}}_{A \otimes B}$ described in step 7 of this work assumes the trace of input events $t = (t_A, t_B)$ to be serializable and provided as a sequence of events in $\mathcal{E}^-_{A\otimes B}$ ($\dot{\mathcal{R}}_{A\otimes B} : \Sigma_{A\otimes B} \times \mathcal{E}^-_{A\otimes B} \to \Sigma_{A\otimes B}$).

Note that we can relax the assumption on the serializable traces because any trace of events $t_{A\otimes B} = (t_A, t_B)$ in $A \otimes B$ can be seen as 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 viceversa and such that $(t_{1A} :: t_{2A} :: .. :: t_{nA}) = t_A$ and $(t_{1B} :: t_{2B} :: .. :: t_{nB}) = t_B$. Then, for each t_i,

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

$\mathcal{R}_{A\otimes B}$ is a sound monitor that is no less precise than $\hat{\mathcal{R}}_(\sigma, t)$ and does not require t to be serializable. In the next section, we illustrate an example of how precision of the tracking can be further increased by aggregating information from different layers.

5.2 Step 5: Increasing precision — Example

Consider an application loading file f and two monitors, one for the application (A) and one for the operating system (B), both sound w.r.t. $\perp$. This generates the trace $t = \langle \texttt{LOAD}_s\texttt{(}f\texttt{)}, \texttt{OPEN(}f, fd\texttt{)}, \texttt{READ(}fd\texttt{)}, \texttt{CLOSE(}fd\texttt{)}, \texttt{LOAD}_e\texttt{(}f\texttt{)} \rangle$ where the first and last events happen at layer A and all the others at layer B (Fig. 3).

Because files are not properly modeled in A, the source of the transfer in A is given by c^U_A (see definition of c^U in § 2). Because the file is unknown to the application, 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 $t|_A$ induces then a flow of all data $\mathcal{D}$ from c^U_A to c_A, where c_A is an internal container of the application, e.g. a document.

At the OS level, the file has a proper abstraction. Let file_f be such a container and d the data item stored in it. The execution of $t|_B$ is then modeled in B as a flow from file_f to container m_{app} representing the memory of the application.

If A and B were considered in isolation, the storage of c_A and m_{app} after the execution of t would be, respectively, $\mathcal{D}$ and d. Using the model presented in Step 4, instead, both containers would contain $\mathcal{D}$, a sound but coarse approximation.

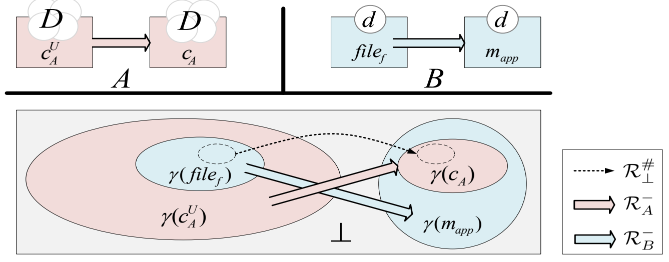

Fig. 3. Example of application loading a file, according to single layer monitors. Dotted sets represent actual $\mathbb{S}_{\mathcal{R}^\#_\perp}$ and $\mathbb{D}_{\mathcal{R}^\#_\perp}$ sets.

A better approximation can be provided by observing that $\gamma_B(\textit{file } f) \subseteq \gamma_A(c_A^U)$ and $\gamma_A(c_A) \subseteq \gamma_B(m_{app})$, 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, the content of all the $\perp$-containers in $\gamma_B(m_{app})$ after the execution of t, including those in $\gamma_A(c_A)$, is at most d, as reported by $\mathcal{R}_B^-$ and because of its soundness. Therefore, a more precise monitor for the combined system would model t as a flow from $\gamma_B(\textit{file } f)$ to $\gamma_A(c_A)$, thus estimating that after the execution both c_A and m_{app} contain d. Note that this result is more precise than $\mathcal{R}_A^-$'s estimation.

What this scenario illustrates is that one layer has a more precise knowledge than the other about *the sources* of a certain event (e.g. the content of the file), while the other layer has a finer-grained understanding of the *destination* of the transfer (e.g. the app-specific container c_A). Let the term *cross actions* indicate those high-level operations, like "`'Application x loading file` *f*`'`" in the example (cf. Fig. 4), that correspond to traces of events at both layers in which this intuition holds.

In the following, we characterize events in this kind of traces, by referring to `IN`,`OUT` and `INTRA` as *behaviors* of events. If a certain cross action generates two events $e_A \in \mathcal{E}_A^-$ and $e_B \in \mathcal{E}_B^-$ such that $\gamma_A(\mathbb{S}_{\mathcal{R}_A^-}(e_A)) \subseteq \gamma_B(\mathbb{S}_{\mathcal{R}_B^-}(e_B))$ and $\gamma_B(\mathbb{D}_{\mathcal{R}_B^-}(e_B)) \subseteq \gamma_A(\mathbb{D}_{\mathcal{R}_A^-}(e_A))$, we say that e_A is an `OUT` event and that e_B is an `IN` event. 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 $\dagger$ propagates data within $\dagger$ according to $\mathcal{R}_\dagger^-$ and, in turn, to any other layer via related containers.

Hence, in addition to the dependency between layers generated by related containers and discussed in § 4.2, we consider also a second class of cross-layer flows, i.e. those due to `IN` and `OUT` events.

Definition 9. *A* cross-layer flow *of data is generated by either: (1) the result of executing an event that transfers data to a container at one layer that is related with a container at the other layer, or (2) 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 flowed to the destinations of a certain `IN` event (e.g. `LOAD()`) is at most the same data read by the respective `OUT` events (e.g. `READ()`). In the next subsections we capture the two kinds of cross-layer dependencies described in Definition 9 in form of two oracles, which describe the relation between two layers. Provided an instantiation of these oracles, and the models for the layers, it is possible to automatically generate a sound precise model for the whole system composed by both layers (§ 5.4).

5.3 Step 6: Oracles definition

X_A *oracle:* Information about related container, as needed by the model described in Step 4, can be captured by oracle $X_A : \mathcal{C}_{A\otimes B} \to \mathbb{P}(\mathcal{C}_{A\otimes B})$, which maps each container c to the set of all the containers related to c at other layers.

Oracle Property 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, $sync : \Sigma_{A\otimes B} \to \Sigma_{A\otimes B}$ returns a new state in which all the data stored in each container have been propagated to all the related containers at other layers, i.e. $\forall c \in \mathcal{C}_{A\otimes B}, \sigma \in \Sigma_{A\otimes B} \bullet sync(\sigma)(c) = \sigma(c) \cup \bigcup_{c' \in X_A(c)} \sigma(c')$. Because the *sync* operator only adds data to containers, it is easy to prove that if σ is a sound state (cf. § 3.2), then $\sigma' = sync(\sigma)$ is also a sound state.

X_B *oracle:* In a multi-layer system, the behavior of a given event may differ in different contexts. For instance, 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 the application's one and if the target file of the system call is the same file being loaded by the application. Similarly, if the application is loading two files at the same time, then a sound and precise modeling needs to associate each `LOAD()` with the respective `READ()` events only.

To model this distinction, we use a unique identifier, called *scope id*, for each distinct instance of a cross action. All the `IN` and `OUT` events at both layers that pertain to a certain cross action are associated to that cross action's scope id.

This is captured by oracle $X_B : \mathcal{E}^{-}_{A\otimes B} \times \Sigma \to \{\mathtt{IN}, \mathtt{OUT}, \mathtt{INTRA}\} \times SCOPE$, where $SCOPE$ is the set of scope ids, like `'Application x loading file f'`. X_B maps each event to its respective behavior in the context of a cross action.

It is also important to aggregate and store the content of the data being transferred by the `OUT` events in a way that is usable by the next corresponding `IN` event, because multiple `IN`(`OUT`) events may correspond to the same `OUT`(`IN`) event, e.g. one `LOAD()` event may correspond to multiple `READ()` system calls.

For each scope id sc, we model the existence of an *intermediate* container c_{sc} for the cross layer flow. Storage information for the intermediate containers ($\mathcal{C}_{sc}$) must be 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. We model the flow from c_s to c_d 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. For this reason, in this work we consider only *serialized* traces, (i.e. where the sorting of indexed events by timestamp is unique, cf. Definition 8), and where `IN` events take place *after* the respective `OUT` events. This assumption is not restrictive in practice and always held in concrete instantiations [16,15].

In summary, augmenting the set of states for the composed system $\Sigma_{A\otimes B} \subseteq \Sigma_A \times \Sigma_B \times (\mathcal{C}_{sc} \to \mathbb{P}(\mathcal{D}))$, we can encode the relation between two given layers

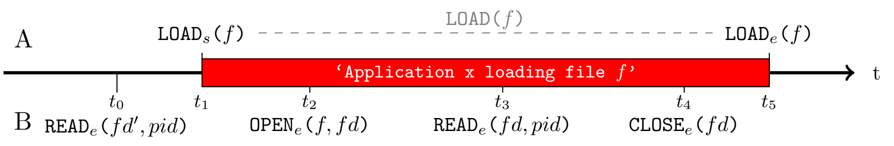

Fig. 4. Example of cross action. The X_B oracle applied to generic $\texttt{READ}_e\texttt{(..)}$ events at time t_0 returns $(\texttt{INTRA}, \emptyset)$. The $\texttt{READ}_e\texttt{(..)}$ event at time t_3 instead, being part of the loading cross action, corresponds to $(\texttt{OUT}, \texttt{'Application x loading file } f\texttt{'})$. The respective `IN` event at layer A is the $\texttt{LOAD}_e\texttt{(..)}$ event at time t_5.

A and B by using the oracles $X_A : C_{A\otimes B} \to 2^{C_{A\otimes B}}$ and $X_B : \mathcal{E}^-_{A\otimes B} \times \Sigma_{A\otimes B} \to \{\texttt{IN}, \texttt{OUT}, \texttt{INTRA}\} \times \mathit{SCOPE}$, which, by definition, guarantee the following:

Oracle Property 2 *Let $t \in seq(\mathcal{E}^-_{A\otimes B})$ be a trace of events terminating with the event e^i, identified as* `IN` *by the oracle X_B. Let $E^O \subseteq \mathcal{E}^-_{A\otimes B}$ be a set of respective (i.e. w.r.t. the same scope) events in t identified as* `OUT` *by the oracle X_B. Then, in an ideal monitoring ($\mathcal{R}^{\#}_{A\otimes B}$) of t, the destinations of e^i contains at most the content of the sources of all the events in E^O.*

Formally, 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, then, guarantee that

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

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

The intuition here 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 past `OUT` events. This is the key behind the refined precision offered by $\dot{\mathcal{R}}_{A\otimes B}$ in comparison to $\hat{\mathcal{R}}_{A\otimes B}$.

5.4 Step 7: Algorithm for sound composition

We now come to the main result of this paper. Our goal is to show that, given an instantiation of the oracles for which the two properties defined in § 5.3 hold, a composition algorithm considering such oracles is sound w.r.t. an ideal monitor at $\perp$, and thus to weak-secrecy.

Let $\gamma_{A\otimes B}$ be the overloading of γ for $C_{A\otimes B}$, $\Sigma_{A\otimes B}$, $\mathcal{E}^-_{A\otimes B}$ and traces of events in $\mathcal{E}^-_{A\otimes B}$. Given the models for A and B and these two oracles, the model $A \otimes B$ for the composed system is specified as follows: First, the set of containers in the system $\mathcal{C}_{A\otimes B}$ is given by $\mathcal{C}_A \cup \mathcal{C}_B \cup \mathcal{C}_{sc}$, where $\mathcal{C}_{sc}$ is the set of intermediate containers (which represent no real container in the system, i.e.

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

```
1  $s_{sc_{RET}} \longleftarrow s_{sc}$; $\sigma_{A_{RET}} \longleftarrow \sigma_A$; $\sigma_{B_{RET}} \longleftarrow \sigma_B$;
2  $(beh, sc) \longleftarrow X_B((\sigma_A, \sigma_B, s_{sc}), e)$;
3  switch $beh$ do
4      case INTRA
5          if $e \in \mathcal{E}_A^-$ then $\sigma_{A_{RET}} \longleftarrow \mathcal{R}_A^-(\sigma_A, e)$;
6          else $\sigma_{B_{RET}} \longleftarrow \mathcal{R}_B^-(\sigma_B, e)$;
7      case IN
8          if $e \in \mathcal{E}_A^-$ then $\sigma_{A_{RET}} \longleftarrow (\sigma_A[t \leftarrow \sigma_A(t) \cup s_{sc}(c_{sc})]_{t \in \mathbb{D}_{\mathcal{R}_A^-}(e)})$;
9          else $\sigma_{B_{RET}} \longleftarrow (\sigma_B[t \leftarrow \sigma(t) \cup s_{sc}(c_{sc})]_{t \in \mathbb{D}_{\mathcal{R}_A^-}(e)})$;
10     case OUT
11         if $e \in \mathcal{E}_A^-$ then
12             $s_{sc_{RET}} \longleftarrow s_{sc}[c_{sc} \leftarrow \sigma_A(t)]_{t \in \mathbb{S}_{\mathcal{R}_A^-}(e)}$;
13             $\sigma_{A_{RET}} \longleftarrow \mathcal{R}_A^-(\sigma_A, e)$;
14         else
15             $s_{sc_{RET}} \longleftarrow s_{sc}[c_{sc} \leftarrow \sigma_B(t)]_{t \in \mathbb{S}_{\mathcal{R}_A^-}(e)}$;
16             $\sigma_{B_{RET}} \longleftarrow \mathcal{R}_B^-(\sigma_B, e)$;
17 return $sync(\sigma_{A_{RET}}, \sigma_{B_{RET}}, s_{sc_{RET}})$
```

$\forall c \in \mathcal{C}_{sc} \bullet \gamma_{A\otimes B}(c) = \emptyset$). Secondly, 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 and the storage function for intermediate containers s_{sc}, $\sigma = (\sigma_A, \sigma_B, s_{sc})$.

Given two sound instantiations of the model for A and B and the two oracles defined above, a sound and precise model of the data flows within and across these two layers is captured by $\dot{\mathcal{R}}_{A\otimes B}$ defined in Algorithm 1[1] .

Theorem 2. *Given two oracles X_A and X_B, for which properties 1 and 2 hold, two monitors for two layers $\mathcal{R}_A^-$, $\mathcal{R}_B^-$, an initial state $\sigma_{A\otimes B} = (\sigma_A, \sigma_B)$ and a serializable trace of events $t \in seq(\mathcal{E}_{A\otimes B}^-)$, if $\sigma_\perp \vdash \sigma_A$, $\sigma_\perp \vdash \sigma_B$, $\mathcal{R}_\perp \vdash \mathcal{R}_A^-$ and $\mathcal{R}_\perp \vdash \mathcal{R}_B^-$, then $\dot{\mathcal{R}}_{A\otimes B}((\sigma_A, \sigma_B), (t_A, t_B))$ is sound, i.e. $\mathcal{R}_{A\otimes B}^{\#} \vdash \dot{\mathcal{R}}_{A\otimes B}$.*

Proof. We provide an inductive argument for the soundness of our approach over the length of the trace.

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 (line 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 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

[1] Let m be a function of type $S \to T$ and $X \subseteq S$. $m' = m[x \leftarrow expr]_{x\in X}$ indicates a function $S \to T$ such that $m'(y) = expr$ for any $y \in X$ and $m'(y) = m(y)$ otherwise.

kind of flows is captured by the *sync* operation (line 17), 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 13) (sound estimation thanks to single layer soundness) and to an intermediate container (line 12). Because intermediate containers have no concretization in $\perp$, their content is irrelevant for soundness check; we thus end up in the same state as for the INTRA event and, after syncing the content with the containers at the other layer (line 17), confirm the soundness using the same argument.

The case of e being an IN event is not contemplated 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, cf. § 5.2).

Inductive Case. Given $t = t^p :: e$ 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, we show that the state $\sigma' = \dot{\mathcal{R}}_{A\otimes B}(\sigma, e)$ is also sound. As before, we assume e to be an event at level A (the case for B is analogous).

As in the previous case, the set of opened scopes is used only to decide the cross-layer behavior of e and is sound by the oracle assumption. Similarly as before, 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, being an OUT event equivalent to an INTRA event and being the intermediate containers 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 8).

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 12). 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 oracle property 2 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 8) results in a sound state.

In more details, it is clear from the algorithm (line 8) 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 $d \notin \dot{\mathcal{R}}_{A\otimes B}(\sigma, e)(c)$. Being e 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 8). 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 oracle property 2, there must exists an event $e^o \in E^O$ in the trace, such that $d \in \sigma_\perp^O(\gamma_B(\mathbb{S}_{\mathcal{R}_B^-}(e^O)))$, where σ^O is the state of the system right before the execution of e^O and $\sigma_\perp^O$ is its concretization.
5. Being σ^O a state of the system reachable by a subtrace of t, $\sigma_\perp^O \vdash \sigma^O$ by inductive hypothesis. Such soundness implies that if $d \in \sigma_\perp^O(\gamma_B(\mathbb{S}_{\mathcal{R}_B^-}(e^O)))$, 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 12). In particular, because the content of intermediate containers is never erased (multiple repetitions of the same cross-layer events 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.

□

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 oracle property 2.

6 Related Work

In terms of system-wide data flow tracking, we distinguish three classes of solutions in the literature. The first class includes solutions that focus on a single layer, like the operating system [10,12,8], the hypervisor [26] or the hardware level [4]. With respect to our model, hardware level solutions could be seen as the the $\perp$ layer. Despite recent improvements in efficiency both at the software [1] and hardware level [6], solutions in this class fail to capture the high-level semantics of events and objects (e.g. "forward a mail").

The second class of solutions includes those approaches that consider multiple instantiations of the same solution for *one* specific level of abstraction, usually the application layer. This class of work includes solutions like [11] and [19], where the inter-application flow tracking relies on the simultaneous execution of the sender and the receiver events, both at the application layer. None of them can model a flow of data toward resources at different layers, e.g toward a file; given a monitor for the second layer, this is instead possible with our model.

The third class of related work includes approaches that consider multiple layers of abstractions at the same time. [17] is a work from the area of provenance aware storage systems, where representations of data are considered at three system layers at the same time (network, file system, workflow engine). Depending on the type of the content being handled, this work relies on tracking solutions that interact with each other and exchange taint results across different layers. Similarly, the *Garm* tool [7], aims at tracking data provenance information across multiple applications and machines. Garm instruments application binaries to track and store the data flow within and across applications, and to monitor interactions with the OS. Although both [17] and [7] address multiple layers of abstraction at the same time, none describes a general model applicable to a different number or type of layers, but each rather focuses on hard-coded solutions for the specific layers of abstraction considered.

[18] addresses multiple layers of abstraction generically by integrating a basic data flow tracking schema with a usage control framework. Here, the specification of the cross-layer dependencies is performed ad-hoc and all the monitors are executed in parallel in an independent fashion. Step 6 of [18] defines the meaning of cross-layer flows at the semantics model, but does not provide any notion of soundness, nor any operationalized way to monitor such flows at runtime.

A work more related to ours is *Shrift* [16], a solution for system-wide hybrid information flow tracking. Shrift replaces the runtime monitoring of an application with a statically computed mapping between its inputs and outputs, which is used at runtime by an operating system layer monitor to model data flows through the application. While using the model presented here, [16] does not describe cross layer flows in general.

7 Conclusions

In this paper we presented a formal definition of soundness, in terms of a relaxed notion of non-interference (weak-secrecy), for system-wide data flow tracking at and across different layers of abstraction. This semantic characterization of soundness is the first of its kind and represents the paper's first contribution.

We also proposed a generic schema to compose data flow analyses at various levels. Our schema relies on the existence of partial oracles that spell out the relation between the different levels in an actual system. The operationalization of the composition as an algorithm for runtime monitoring and the proof of its soundness represent the second major contribution of this research.

It is crucial to make the oracle assumptions explicit, even though in practice it is challenging to prove that single layer monitors and oracles are accurate, due for instance to non-deterministic low level interleavings and implementation details such as temporary variables and files. Such assumptions are usually reasonable, given that domain experts can accurately model the data-flow propagation of single high-level events, and whenever relations between layers are well known.

We have instantiated the framework described in this work to connect instantiations for different layers of abstraction, including a mail client [14], X86 binaries [2], Java Bytecode [16], and different operating systems [24,10], proving its feasibility. We argue that the genericity of the approach makes it possible to capture other solutions for data flow tracking from the literature, e.g. [27,5,3], as single-layer monitor instances, and to connect them to trackers at other layers in a sound manner. We do not discuss implementation details and experiments here, but refer to [15] for more information.

In sum, our proposed cross-layer algorithm $\dot{\mathcal{R}}_{A\otimes B}$ conservatively estimates and synchronizes the data propagation state between layers or inside one layer given that monitors are sound in isolation, that oracles are accurate, and that traces are serializable. Our implementation experiments show that these conditions are met often in practice, allowing for a sound and precise analysis. If, however, some of these conditions are not met, then one is forced to use a more conservative analysis (like $\hat{\mathcal{R}}_{A\otimes B}$) which propagates data from all sources to all destinations of a trace. Ultimately, if sources and destinations are unknown, the only possible sound analysis is to propagate all data to all containers

References

1. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. *ACM Sigplan Notices*, 44(8):20–31, 2009.
2. A. K. Biswas. Towards improving data driven usage control precision with intra-process data flow tracking. Master's thesis, Technische Universität München, 2014.
3. E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, pages 3–12, 2009.
4. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, 2004.
5. J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings MICRO37*, pages 221–232. IEEE, 2004.
6. A. A. de Amorim, M. Dénes, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies. 2015.
7. B. Demsky. Cross-application data provenance and policy enforcement. *ACM Transactions on Information and System Security*, 14(1):1–22, May 2011.
8. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *USENIX OSDI*, 2010.
9. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, 1982.
10. M. Harvan and A. Pretschner. State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *NSS*, 2009.
11. H. C. Kim, A. D. Keromytis, M. Covington, and R. Sahita. Capturing information flow with concatenated dynamic taint analysis. In *ARES*, 2009.
12. M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. *SOSP*, 2007.

13. P. Kumari, A. Pretschner, J. Peschla, and J.-M. 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. ACM, 2011.
14. M. Lörscher. *Usage Control for a Mail Client.* Master thesis, TU Kaiserslautern, 2012.
15. E. Lovat. *Cross-layer Data-centric Usage Control.* PhD thesis, Technische Univesität München, 2015.
16. E. Lovat, A. Fromm, M. Mohr, and A. Pretschner. SHRIFT System-wide HybRid Information Flow Tracking. In *IFIP SEC 2015 (to appear)*. 2015.
17. K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. USENIX, 2009.
18. A. Pretschner, E. Lovat, and M. Büchler. Representation-independent data usage control. In *DPM/SETOP*, pages 122–140, 2011.
19. S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *ARES*, 2014.
20. G. Smith. On the foundations of quantitative information flow. In *Foundations of software science and computational structures*, pages 288–302. Springer, 2009.
21. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM SIGARCH*, 2004.
22. D. Volpano. Safety versus secrecy, 1999.
23. D. Volpano and G. Smith. *A type-based approach to program security.* Springer, 1997.
24. T. Wüchner and A. 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.
25. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *CCS*, 2007.
26. Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: System support for derived data management. *SIGPLAN Not.*, 45(7):63–74, Mar. 2010.
27. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical Report UCB/EECS-2009-145, EECS Department, University of California, Berkeley, Oct 2009.