End-to-end Multilevel Hybrid Information Flow Control*

Lennart Beringer

Department of Computer Science, Princeton University,
35 Olden Street, Princeton NJ 08540
eberinge@cs.princeton.edu

Abstract. We present models and soundness results for hybrid information flow, i.e. for mechanisms that enforce noninterference-style security guarantees using a combination of static analysis and dynamic taint tracking. Our analysis has the following characteristics: (i) we formulate hybrid information flow as an end-to-end property, in contrast to disruptive monitors that prematurely terminate or otherwise alter an execution upon detecting a potentially illicit flow; (ii) our security notions capture the increased precision that is gained when static analysis is combined with dynamic enforcement; (iii) we introduce path tracking to incorporate a form of termination-sensitivity, and (iv) develop a novel variant of purely dynamic tracking that *ignores* indirect flows; (v) our work has been formally verified, by a comprehensive representation in the theorem prover Coq.

1 Introduction

Hybrid information flow techniques integrate static analyses with dynamic taint tracking or execution monitoring to ensure the absence of illicit flows. In the systems community, instrumentations that control *direct* data flows in assignments have been refined, using compile-time transformations or dynamic binary instrumentation, to capture *indirect* (or *implicit*) flows arising from the branching behavior or aliasing [29,23,14,19,18]. Typically, these systems focus on efficient implementability and are justified by informal arguments, but lack mathematically precise definitions of the intended security guarantee or formal soundness proofs.

The inverse observation applies to work on language-based security. Here, the traditional focus on static techniques has recently been complemented by studies of coupled or inlined monitors, with different mechanisms for detecting, preventing, or handling potentially illicit flows [30,28,27,13,24,20,3,4,22]. Typically, these analyses are illustrated with proof-of-concept implementations that are less comprehensive than those of the systems community but are backed up with (pencil-and-paper) soundness proofs based on precise definitions of operational models and the intended security guarantee.

Our long-term goal is to integrate flexible information flow enforcement into frameworks such as the Verified Software Toolchain [2]. As a stepping stone, and building on formalizations of type systems for noninterference [7,10,1], we present the first hybrid enforcement that is backed up by an implementation in a proof assistant, Coq [9].

* This work was funded in part by the Air Force Office of Scientific Research (FA9550-09-1-0138).

R. Jhala and A. Igarashi (Eds.): APLAS 2012, LNCS 7705, pp. 50–65, 2012.

Our analysis takes inspiration from RIFLE [29], one of the first hybrid systems with a precise treatment of indirect flows. In particular, we provide an analysis of one of RIFLE's core ideas, the use of separate taint registers for direct and indirect flows. Underpinning RIFLE's intuitive soundness argument with a formal guarantee, we show in which sense the tracking of indirect flows using dedicated **join** instructions improves precision when compared to a more naive system where only data taints are tracked dynamically. The latter system in turn is more precise than the standard static system, the flow-sensitive type system of Hunt and Sands [16,17].

The guarantees recently proposed for inlined [13] or non-inlined [27] monitors prematurely terminate or alter executions upon detecting a potentially illicit flow. Our analysis shows that dynamic enforcement may alternatively be understood as an *asymmetric* end-to-end indistinguishability notion that refines (multilevel) noninterference. Here, the asymmetry captures the difference between the actual (taint-instrumented) execution and hypothetical competitor executions (which may be taint-instrumented or not). The resulting notion captures the systems-oriented intuition that each value's final taint should carry fine-grained information regarding its (potential) origins.

Asymmetric interpretations have previously been considered by Le Guernic and Jensen [21,20], and also by Magazinius et al. [22]. These works limit their attention to two-level security and track direct and indirect flows jointly. In addition, Le Guernic and Jensen specify the set of observable (final) variables statically. Our analysis exposes additional fine-grained structure of taint tracking and highlights that the noninfluence between taints and the native execution actually refines to a ternary discipline: data taints (capturing direct flows) are unaffected by control taints (capturing indirect flows), and neither one affects the data plane (i.e. the native execution).

Finally, we introduce *path tracking*, by adding a further taint register that collects the taints of all branch conditions encountered. Path tracking propagates termination from tracked to untracked executions, and also provides a meaningful indistinguishability guarantee for pure data flow tracking. In particular, we outline in which sense control taints can be safely eliminated without detrimentally affecting the security guarantee.

Summarizing our contributions, we present an analysis of hybrid information flow enforcement that considers fine-grained taint-tracking with separated control and data taints. We develop appropriate notions of formal security, prove corresponding soundness results, and investigate the relative precision of different syntheses. We introduce path tracking as an extension of previous instrumentations, obtaining termination-aware tracking and an (to our knowledge: the first) extensional interpretation of data flow tracking. In contrast to all previous developments of hybrid information flow or dynamic flow tracking, our development is backed up by a formalization in Coq – see [9].

In contrast to RIFLE's assembly-level formulation, our analysis is carried out for the language of structured commands and loops. This setting suffices for studying the intended security notion and the core aspects of **join**-based taint instrumentation, but a study of additional aspects would certainly be profitable. Of particular interest in this regard would be RIFLE's treatment of memory, which combines a reaching-definitions analysis with external aliasing analyses. However, the exploitation of aliasing information in RIFLE's synthesis is such that use-sites of taint registers are not necessarily dominated by their def-sites, prohibiting a proof of soundness along the program

structure. For this reason, our synthesis uses a different allocation policy for taint registers. A future integration of memory may allow us to study this aspect in more detail and may take additional inspiration from the recent work of Moore and Chong [24].

2 Static Flow-Sensitive Information Flow

We start by fixing some notation and revising aspects of Hunt & Sands' static analysis.

2.1 Native Language

For disjoint sets $\mathcal{X}$ of program variables and $\mathcal{V}$ of value constants, our language concerns expressions $\mathcal{E}$ and commands $\mathcal{C}$ according to the grammar

$$e \in \mathcal{E} ::= x \mid v \mid e \oplus e$$
$$C \in \mathcal{C} ::= \mathbf{skip} \mid x{:=}e \mid C; C \mid \mathbf{if}\ e\ \mathbf{then}\ C\ \mathbf{else}\ C \mid \mathbf{while}\ e\ \mathbf{do}\ C$$

where $x, y, \ldots \in \mathcal{X}$, $v, w, \ldots \in \mathcal{V}$ and $\oplus$ ranges over binary operations. The set of program variables possibly modified (i.e. assigned to) in some command C is denoted by $\mathsf{MV}(C)$. Expression evaluation $s \vdash e \Downarrow v$ and (big-step) command execution $s \xrightarrow{C} t$ are formulated over stores $s, t, \ldots$ from the space $\mathcal{S} = \mathcal{X} \to \mathcal{V}$. Throughout the paper, update operations of various total or partial functions are written as $.[. \mapsto .]$, and lookups as $.(.)$ or simply juxtaposition. We denote empty lists (over various types) by ϵ, list append by @, and list prefixing by ::. The definition of these auxiliary notions, and of the operational judgements are entirely standard and hence omitted.

We formulate our analysis generically over a semilattice $\mathcal{L}$ with partial order $\sqsubseteq$: $\mathcal{L} \to \mathcal{L} \to \mathsf{Prop}$, least upper bound $\sqcup : \mathcal{L} \to \mathcal{L} \to \mathcal{L}$, and bottom element $\bot$. We typically write the binary operators in infix position and extend $\sqcup$ to subsets of $\mathcal{L}$, with $\sqcup \varnothing = \bot$, and also to $\mathcal{L}^*$ (i.e. finite lists over $\mathcal{L}$), with $\sqcup \epsilon = \bot$. Example programs use the ternary semilattice $\mathit{low} \sqsubsetneq \mathit{mid} \sqsubsetneq \mathit{high}$ and silently name variables according to their typical *initial* taint: $l : \mathit{low}, m : \mathit{mid}, h : \mathit{high}$. Of course, the static level and the dynamic taint of a variable may differ from this at different program points.

2.2 Flow-Sensitive Type System in the Style of Hunt and Sands

The starting point of our analysis is the flow-sensitive type system for multilevel noninterference by Hunt & Sands [16,17]. Its algorithmic formulation (given in [17]) employs judgements $\vdash \{\Gamma\}C\{\Delta\}$ where the contexts Γ, Δ associate security elements to all program variables and also to a distinguished pseudo-variable $\mathbf{pc} \notin \mathcal{X}$. We denote the restriction of context Γ to the program variables by $|\Gamma|$ and let $G, H, \ldots$ range over such $\mathbf{pc}$-erased contexts. Lattice constants and operations are lifted to erased or unerased contexts in the standard pointwise fashion. Label evaluation $[\![e]\!]_G$ is given by

$$[\![x]\!]_G = G(x) \qquad [\![v]\!]_G = \bot \qquad [\![e_1 \oplus e_2]\!]_G = [\![e_1]\!]_G \sqcup [\![e_2]\!]_G$$

and is extended to non-erased contexts via $[\![e]\!]_\Gamma = [\![e]\!]_{|\Gamma|}$. The proof rules for $\vdash \{\Gamma\}C\{\Delta\}$ are summarized in Figure 1, with an explicit construction of the (least) fixed point context in rule HS-WHILE. We write $\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$ to refer to a particular derivation for $\vdash \{\Gamma\}C\{\Delta\}$. It is easy to see that $\vdash \{\Gamma\}C\{\Delta\}$ implies

$$\text{HS-Comp}\ \frac{\forall i \in \{1,2\}.\ \vdash \{\Gamma_i\}C_i\{\Gamma_{i+1}\}}{\vdash \{\Gamma_1\}C_1;C_2\{\Gamma_3\}} \qquad \text{HS-Ass}\ \frac{\Delta = \Gamma[x \mapsto \Gamma(\mathbf{pc}) \sqcup [\![e]\!]_\Gamma]}{\vdash \{\Gamma\}x{:=}e\{\Delta\}}$$

$$\text{HS-Skip}\ \frac{}{\vdash \{\Gamma\}\mathbf{skip}\{\Gamma\}} \qquad \text{HS-Ite}\ \frac{\begin{array}{c} p = \Gamma(\mathbf{pc}) \qquad \Gamma' = \Gamma[\mathbf{pc} \mapsto p \sqcup [\![e]\!]_\Gamma] \\ \forall i \in \{1,2\}.\ \vdash \{\Gamma'\}C_i\{\Delta_i\} \end{array}}{\vdash \{\Gamma\}\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\{(\Delta_1 \sqcup \Delta_2)[\mathbf{pc} \mapsto p]\}}$$

$$\text{HS-While}\ \frac{\begin{array}{c} \forall\, i < n.\ \Gamma_i \neq \Gamma_{i+1} \qquad \forall\, i \geqslant n.\ \Gamma_i = \Gamma_{i+1} \qquad p = \Gamma(\mathbf{pc}) \\ \Gamma_0 = \Gamma[\mathbf{pc} \mapsto p \sqcup [\![e]\!]_\Gamma] \qquad \forall i.\ \vdash \{\Gamma_i\}C\{\Delta_i\} \\ \forall i.\ \Gamma_{i+1} = (\Delta_i \sqcup \Gamma)[\mathbf{pc} \mapsto (\Delta_i \sqcup \Gamma)(\mathbf{pc}) \sqcup [\![e]\!]_{\Delta_i \sqcup \Gamma}] \end{array}}{\vdash \{\Gamma\}\mathbf{while}\ e\ \mathbf{do}\ C\{\Gamma_n[\mathbf{pc} \mapsto p]\}}$$

Fig. 1. System $\vdash \{\Gamma\}C\{\Delta\}$ by Hunt & Sands with explicit (least) fixed point in rule HS-While

- $\Gamma(\mathbf{pc}) = \Delta(\mathbf{pc})$
- $\Delta = \Delta'$ whenever $\vdash \{\Gamma\}C\{\Delta'\}$ (functionality of H. & S.-typing)
- $\Delta' \sqsubseteq \Delta$ whenever $\vdash \{\Gamma'\}C\{\Delta'\}$ and $\Gamma' \sqsubseteq \Gamma$ (monotonicity of H. & S.-typing)
- $\Delta(\mathbf{pc}) \sqsubseteq \Delta x$ for $x \in \mathsf{MV}(C)$, and $\Delta x = \Gamma x$ for $x \notin \mathsf{MV}(C)$.

Furthermore, we have the following:

Lemma 1. *For* $\vdash \{\Gamma\}C\{\Delta\}$ *and* $s \xrightarrow{C} t$, *any* $x \in \mathcal{X}$ *satisfies* $\Delta(\mathbf{pc}) \sqsubseteq \Delta x$ *or* $sx = tx \wedge \Gamma x \sqsubseteq \Delta x$.

Following common practice, our security notions are formulated using *indistinguishability* relations over stores, given some threshold $\kappa \in \mathcal{L}$:

Definition 1. *States* s *and* s' *are* G*-indistinguishable below* $\kappa \in \mathcal{L}$, *notation* $s =^G_{\sqsubseteq\kappa} s'$, *if all* x *with* $Gx \sqsubseteq \kappa$ *satisfy* $sx = s'x$. *Command* C *is* κ*-secure for* G *and* H, *notation* $\models_\kappa \{G\}C\{H\}$, *if* $s =^G_{\sqsubseteq\kappa} s'$ *implies* $t =^H_{\sqsubseteq\kappa} t'$ *whenever* $s \xrightarrow{C} t$ *and* $s' \xrightarrow{C} t'$.

We note that for $\kappa' \sqsubseteq \kappa$ and $G \sqsubseteq G'$, $s =^G_{\sqsubseteq\kappa} s'$ implies $s =^{G'}_{\sqsubseteq\kappa'} s'$ (monotonicity of indistinguishability). Soundness of $\vdash \{\Gamma\}C\{\Delta\}$ is then given by the following result.

Theorem 1. *If* $\vdash \{\Gamma\}C\{\Delta\}$ *then* $\models_\kappa \{|\Gamma|\}C\{|\Delta|\}$ *for any* κ.

3 Data Flow Tracking

The first step towards a more dynamic regime is a language where only direct flows (from e into x in assignments $x{:=}e$) are tracked. Indirect flows are treated by program annotations based on the static analysis, so that the taints of all affected assignments are incremented by the branch conditions' *static* security levels. While being less permissive than the system we will develop in Section 4, this language suffices for motivating our formulation of soundness and discussing some aspects of the program synthesis.

3.1 Language with Decorated Assignments

The category of taint-instrumented commands $\mathcal{T}$ (typically ranged over by T) agrees with that for native commands $\mathcal{C}$ except that assignments take the form $[\lambda]\, x{:=}e$ where $\lambda \in \mathcal{L}^*$. We denote the native command arising from recursively erasing all decorations from assignments by $|T|$.

Operationally, the taint-extended language manipulates states $\sigma, \tau, \ldots$ that are pairs of stores s and (erased) contexts[1] G. Command evaluation $\sigma \xrightarrow{T}_{\mathsf{T}} \tau$ is defined in parallel to the native $s \xrightarrow{C} t$, with the exception that an instrumented assignment $[\lambda]\, x{:=}e$ additionally updates the taint for x to the lub of λ and the taint of e (given by the lub of the taints associated with the variables in e):

$$\text{T-Ass}\ \frac{s \vdash e \Downarrow v \qquad [\![e]\!]_G = a \qquad \sqcup\, \lambda = l}{(s, G) \xrightarrow{[\lambda]\, x:=e}_{\mathsf{T}} (s[x \mapsto v], G[x \mapsto (l \sqcup a)])}$$

As a consequence, the language satisfies the following simple erasure property: for any G, $s \xrightarrow{|T|} t$ is equivalent to $\exists H.\ (s, G) \xrightarrow{T}_{\mathsf{T}} (t, H)$.

3.2 Program Synthesis

Given a program C, the task of the program synthesis is to generate a program T that is suitably annotated for noninterference and is functionally equivalent to C. To this end, a synthesis must achieve two tasks: first, assignment annotations must be derived that correctly account for all implicit flows. Second, assignments in conditionally executed code regions must be counter-balanced so that no information leaks from the execution *or non-execution* of a particular program path. This is achieved by lifting the taints of all variables *potentially* modified in a piece of code to (at least) the taint of any branch condition enclosing the code fragment, using *compensation code*:

Definition 2. *For $\kappa \in \mathcal{L}$, command C, and an (arbitrary) enumeration $x_1, \ldots, x_n$ of $\mathsf{MV}(C)$, we define $\mathsf{CompCd}_\kappa(C)$ to be $[\kappa]\, x_1{:=}x_1; \ldots; [\kappa]\, x_n{:=}x_n$.*

Thus, $\mathsf{CompCd}_\kappa(C)$ lifts the taints of all x_i to at least κ without changing their data values. Taints already above κ are also unaffected. While the term *compensation code* appears to have been coined by Chudnow and Naumann [13], the concept itself is already present in RIFLE [29] and Venkatakrishnan et al.'s work [30], and represents an extensional, non-disruptive alternative to the policy of *no sensitive-upgrade* [3,5].

The synthesis of annotated code is now defined on the basis of a derivation $\mathcal{D} \vdash \{\Gamma\} C \{\Delta\}$. Figure 2 defines the synthesis of instrumented code $\theta(\mathcal{D}, C)$ by case distinction on C, with recursive reference to the subderivations of $\mathcal{D}$.

The annotation $[\Gamma(\mathbf{pc})]$ in assignments models the indirect flows from enclosing branch conditions into assigned variables, as statically approximated by context Γ. Direct flows from variables in e into x are dealt with (in a dynamically precise manner) by the second side condition of the operational rule T-Ass and thus need not be taken

[1] This treatment is equivalent to annotating values with their taint directly, i.e. to a model with stores that map registers to *tainted* values.

C	$\theta(\mathcal{D}, C)$	where ...
skip	**skip**	
$x{:=}e$	$[\Gamma(\mathbf{pc})]\, x{:=}e$	
$C_1; C_2$	$\theta(\mathcal{D}_1, C_1); \theta(\mathcal{D}_2, C_2)$	$\mathcal{D}_i \vdash \{\Gamma_i\} C_i \{\Gamma_{i+1}\}$, $(\Gamma, \Delta) = (\Gamma_1, \Gamma_3)$
if e **then** C_1 **else** C_2	**if** e **then** $\theta(\mathcal{D}_1, C_1)$ **else** $\theta(\mathcal{D}_2, C_2)$; $\mathsf{CompCd}_\kappa(C_1; C_2)$	$\mathcal{D}_i \vdash \{\Gamma'\} C_i \{\Delta_i\}$ $\kappa = \Gamma(\mathbf{pc}) \sqcup [\![e]\!]_\Gamma$
while e **do** C'	**while** e **do** $\theta(\mathcal{D}_n, C')$; $\mathsf{CompCd}_\kappa(C')$	$\mathcal{D}_n \vdash \{\Gamma_n\} C' \{\Delta_n\}$ $\kappa = \Gamma(\mathbf{pc}) \sqcup [\![e]\!]_{\Gamma_n}$

Fig. 2. Synthesis of taint-instrumented code given a derivation $\mathcal{D} \vdash \{\Gamma\} C \{\Delta\}$. Items are named in accordance with Figure 1: in the cases for composition and conditionals, the derivations $\mathcal{D}_i$ refer to the subderivations for the respective subphrases C_i ($i \in \{1, 2\}$). Similarly in the case for loops: n is the index where the fixed point iteration stabilizes (cf. Figure 1).

into consideration during program generation. The clauses for composition, conditionals, and loops assemble the recursively generated code fragments, adding compensation code in the latter two cases. In the case for loops, it suffices to add compensation code as a loop epilogue. We note that the given taint κ not only dominates $\Gamma(\mathbf{pc}) \sqcup [\![e]\!]_{\Gamma_n}$ but also all $[\![e]\!]_{\Gamma_i}$ ($i < n$), by the monotonicity of typing.

Taint-tracking respects the static analysis in the following sense:

Lemma 2. *For* $\mathcal{D} \vdash \{\Gamma\} C \{\Delta\}$ *and* $(s, G) \xrightarrow{\theta(\mathcal{D}, C)}_{\mathsf{T}} (t, D)$, $G \sqsubseteq |\Gamma|$ *implies* $D \sqsubseteq |\Delta|$.

A typical case where the claim in Lemma 2 holds strictly is the (native) program **if** m **then** $x{:=}3$ **else** $x{:=}h$, where $\Gamma = [h : \mathit{high}, m : \mathit{mid}, \mathbf{pc} : \mathit{low}]$. For $G = |\Gamma|$ we have $Dx = \mathit{mid} \sqsubsetneq \mathit{high} = \Delta x$ whenever s is such that $s \vdash m \Downarrow \mathbf{true}$.

Functional equivalence between $\theta(\mathcal{D}, C)$ and C follows from the above erasure property and the fact that compensation code does not affect the data plane. We now turn our attention to the noninterference guarantee of taint tracking.

3.3 Interpretation and Soundness

Intuitively, the tag associated with a value in a *final* state of a taint-enhanced execution indicates the *initial* values it may have been affected by. Thus, the guarantee is *execution*-oriented rather than *program*-classifying [21]. In contrast to monitor-based formulations that preemptively terminate executions in case of a potential security violation, this intuition treats taints in an end-to-end fashion, but is nontrivial only if at least the instrumented execution terminates. In order to clarify its relationship with noninterference, we give below (Definition 4) a formal reading of our intuition, capturing the asymmetry by designating the tainted (and terminating) execution as the *lead* or *major* execution, and considering the second (tainted or untainted) execution the *minor* or *competitor* execution. Typographically, we distinguish major from minor executions by consistently using primed entities for the latter. Part of the intuitive reading then is that the terminal tags of lead executions determine which minor executions need to be considered, separately for each final value.

As a preliminary step, let us first consider the following symmetric and static notion.

Definition 3. *Program T is* statically (G,H)-secure *if for* $(s,G) \xrightarrow{T}_{\mathsf{T}} (t,D)$ *and* $(s',G) \xrightarrow{T}_{\mathsf{T}} (t',D')$ *and for each* x, $s =^{G}_{\sqsubseteq Hx} s'$ *implies* $tx = t'x$.

By quantifying over x outside of the implication, this notion captures explicitly that final values in x could only be influenced by certain initial values, namely those held in variables y with $Gy \sqsubseteq Hx$ (note that the executions agree on their initial taint state G here). Indeed, the static type system ensures static security:

Theorem 2. *For* $\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$, $\theta(\mathcal{D},C)$ *is statically* $(|\Gamma|,|\Delta|)$*-secure.*

Thus, indistinguishability of s and s' (w.r.t. $|\Gamma|$) below $|\Delta|(x)$ suffices for guaranteeing $tx = t'x$. In fact, it is easy to see that static security reformulates universal κ-security pointwise for each variable:

Lemma 3. *For any* G, H, s, s', t, t'*, the following are equivalent:*

1. *for all* x, $s =^{G}_{\sqsubseteq Hx} s'$ *implies* $tx = t'x$ *(the clause in Definition 3)*
2. *for all* κ, $s =^{G}_{\sqsubseteq \kappa} s'$ *implies* $t =^{H}_{\sqsubseteq \kappa} t'$ *(the clause in Definition 1).*

Applying this lemma to the case where $s \xrightarrow{C} t$ and $s' \xrightarrow{C} t'$ for some C (and using erasure) yields that Theorems 2 and 1 are equivalent - the universal quantification over κ in the latter and the pointwise formulation in Definition 3 are equally powerful.

The effect of the dynamic taints is exploited (and the asymmetry emerges) if we refine Definition 3 as follows, i.e. instantiate H by the dynamic final taint map D:

Definition 4. T *is* dynamically G-secure *if for* $(s,G) \xrightarrow{T}_{\mathsf{T}} (t,D)$ *and* $(s',G) \xrightarrow{T}_{\mathsf{T}} (t',D')$ *and for each* x, $s =^{G}_{\sqsubseteq Dx} s'$ *implies* $tx = t'x$.

Now, the final dynamic taints of the major execution, held in D, determine whether s and s' are indistinguishable, instead of the static $|\Delta|$ as before. As $Dx \sqsubseteq |\Delta|(x)$ holds for each x, this change *relaxes* the condition on competitor states s' to s and hence admits more minor executions for consideration. Indeed, using Lemma 2 and the monotonicity of indistinguishability one may show that dynamic G-security strengthens (i.e. implies) static (G,H)-security, with strict inequality again holding for any variable x for which the type system performs a strictly approximate lub-operation at some control flow merge point (cf. the example at the end of Section 3.2).

For $x \in \mathsf{MV}(C)$, the final taint Dx necessarily dominates the taint of all those branch or loop conditions encountered during the lead execution whose body (whether executed or not) contains an assignment to a variable that (directly or indirectly) flows into x. Hence, these conditionals necessarily evaluate identically in the minor execution.

Like static security, dynamic security may also be expressed using universal quantification over security levels κ: by Lemma 3, Definition 4 is equivalent to the guarantee that for $(s,G) \xrightarrow{T}_{\mathsf{T}} (t,D)$ and $(s',G) \xrightarrow{T}_{\mathsf{T}} (t',D')$, $s =^{G}_{\sqsubseteq \kappa} s'$ implies $t =^{D}_{\sqsubseteq \kappa} t'$, for any κ. In this formulation, the formal asymmetry and the increased precision of dynamic tracking emerge in the final state indistinguishability relation, again via the use of D rather than D' or Δ. Similar comments apply to the results in the remainder of this article: we may always reformulate these results as κ-security for appropriate final-state-indistinguishability relations determined by the final taints of the major execution.

Again, dynamic security is satisfied by synthesized programs:

Theorem 3. *$\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$, $\theta(\mathcal{D}, C)$ is dynamically $|\Gamma|$-secure.*

For the proof of Theorem 3 (and similarly for a direct proof of Theorem 2 that avoids Theorem 1 and Lemma 3), one shows the following generalization, by induction on $\mathcal{D}$. The proofs for conditionals and loops involve case splits on $x \in \mathsf{MV}(C)$, and the proof for loops proceeds by induction on the operational judgement of the lead execution:

Lemma 4. *Suppose $\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$ and $T = \theta(\mathcal{D}, C)$. Let $(s, G) \xrightarrow{T}_{\mathsf{T}} (t, D)$ and $(s', G') \xrightarrow{T}_{\mathsf{T}} (t', D')$, where $G \sqsubseteq |\Gamma|$ and $G' \sqsubseteq |\Gamma|$. Then each x with $\forall y.\ Gy \sqsubseteq Dx \to (sy = s'y \wedge Gy = G'y)$ satisfies $tx = t'x \wedge Dx = D'x$.*

The indistinguishability conditions on the taint components guarantee that no information leaks via the taints themselves. The formulation of Lemma 4 extends the notions used by Magazinius et al. and Le Guernic and Jensen [21,20,22] to multilevel security, and avoids a static classification of variables according to their observation level.

A result similar to Lemma 4 may also be obtained for differently instrumented programs $T = \theta(\mathcal{D}, C)$ and $T' = \theta(\mathcal{D}', C)$ originating from the same native C, where $\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$, $\mathcal{D}' \vdash \{\Gamma'\}C\{\Delta'\}$, $\Gamma \sqsubseteq \Gamma'$, $G \sqsubseteq |\Gamma|$ and $G' \sqsubseteq |\Gamma'|$.

On the other hand, Theorem 3 and the erasure property yield the following relationship between taint-instrumented lead executions and native minor executions:

Corollary 1. *For $\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$ and $G = |\Gamma|$ let $(s, G) \xrightarrow{\theta(\mathcal{D},C)}_{\mathsf{T}} (t, D)$ and $s' \xrightarrow{C} t'$. Then, each x with $s =^{G}_{\sqsubseteq Dx} s'$ satisfies $tx = t'x$.*

In the following section, we will derive a guarantee similar to Corollary 1 for a synthesis that also tracks implicit flows dynamically (Theorem 4).

4 Taint Tracking with Control Dependencies

As discussed above, lowering the pivot κ that governs the indistinguishability of initial states strengthens the guarantee enjoyed by a taint-instrumented program. RIFLE's instrumentation pushes this process further, by replacing the assignment-decorating taint *constants* obtained from typing derivations by taint *variables*, and by complementing them with additional *security registers* that can be explicitly manipulated using a novel instruction **join**. The gain in precision arises from inserting **join**-instructions in such a way that the taint variables are upper-bounded by the constants. In effect, indirect flows are statically converted into additional direct flows that are then tracked dynamically.

In this section we treat a **join**-extended language motivated by RIFLE's insight, but apply a synthesis that avoids assignment annotations. Treating RIFLE's taint variables as a subcategory of security registers, we obtain a slightly simpler model that is closer to the regime of Venkatakrishnan et al. [30]. In order to emphasize their slightly different roles, we track data taints operationally separately from control taints, combining them only when formulating the soundness results.

The development is motivated by the following example.

Example 1. For $\{\kappa, \kappa_y, \kappa_v\} \subseteq \mathcal{L}$ and $\Gamma_0(\mathbf{pc}) \sqsubseteq \kappa$ we have

$$\frac{\vdash \{\Gamma_0\}\mathbf{if}\ e\ \mathbf{then}\ x{:=}m\ \mathbf{else}\ x{:=}h\{\Gamma_1\} \quad \vdash \{\Gamma_1\}\mathbf{if}\ x\ \mathbf{then}\ y{:=}v\ \mathbf{else\ skip}\{\Gamma_2\}}{\vdash \{\Gamma_0\}(\mathbf{if}\ e\ \mathbf{then}\ x{:=}m\ \mathbf{else}\ x{:=}h); \mathbf{if}\ x\ \mathbf{then}\ y{:=}v\ \mathbf{else\ skip}\{\Gamma_2\}}$$

where $\Gamma_0 = [m : mid, h : high, e : \kappa, y : \kappa_y, v : \kappa_v]$, $\Gamma_1 = \Gamma_0[x \mapsto \kappa \sqcup mid \sqcup high]$, and $\Gamma_2 = \Gamma_1[y \mapsto \kappa \sqcup mid \sqcup high \sqcup \kappa_y \sqcup \kappa_v]$. Consequently, synthesis θ annotates the assignment $y{:=}v$ in the second conditional as $[\kappa \sqcup high]\, y{:=}v$, leading to taint level $\kappa_v \sqcup \kappa \sqcup high$ for y whenever this branch is taken. However, for runs from initial states s with $[\![e]\!]_s = \mathbf{true}$ the taint $\kappa \sqcup mid$ for x would suffice, as any competing initial state s' indistinguishable from s below $\kappa \sqcup mid$ necessarily follows the same execution path. Thus, instead of using the *static* level of the branch condition x when annotating $y{:=}v$, we'd prefer to use the *dynamic* one.

In order to replace the use of the *static* level of branch conditions with suitable *dynamic* taints, we introduce a fresh category of *security registers* $\mathcal{Z}$ (typically ranged over by z, with α ranging over $\mathcal{Z}^*$), and extend the category of commands via

$$J \in \mathcal{J} ::= C \mid z{:=}\mathbf{join}\ [e_1, \ldots, e_m]\ [z_1, \ldots, z_n].$$

The language operates over triples $\mu = (s, G, M)$ where $M \in \mathcal{Z} \to \mathcal{L}$ associates lattice elements to security registers. We define the judgement form $\mu \xrightarrow{J}_{\mathsf{J}} \nu$ by embedding the rules of $s \xrightarrow{C} t$ for all instruction forms other than assignments, and adding the rules

$$\text{J-Ass}\ \frac{s \vdash e \Downarrow v \qquad [\![e]\!]_G = a}{(s, G, M) \xrightarrow{x:=e}_{\mathsf{J}} (s[x \mapsto v], G[x \mapsto a], M)}$$

$$\text{J-Join}\ \frac{l = \sqcup_{i=1}^{n} M(z_i) \sqcup \sqcup_{i=1}^{m} [\![e_i]\!]_G \qquad N = M[z \mapsto l]}{(s, G, M) \xrightarrow{z:=\mathbf{join}\ [e_1,\ldots,e_m]\ [z_1,\ldots,z_n]}_{\mathsf{J}} (s, G, N)}$$

Assignments leave the control taints unaffected and update the data taints only based on other data taints. **join**-instructions combine data and control taints to modify the latter but leave the former unchanged. Indeed, it is easy to show formal noninterference results for $\mu \xrightarrow{J}_{\mathsf{J}} \nu$ which express that control taints do not affect data taints and that neither control nor data taints affect the data plane.

4.1 Synthesis

The synthesis of **join**-instrumented programs employs two kinds of security registers. First, for each $x \in \mathcal{X}$, we introduce a security register z^x, intended to hold the implicit flows into x, leaving G to track the direct flows. Second, in order to capture the effect of $\mathbf{pc}$, we introduce security registers z^i ($i \in \mathcal{N}$), which will be allocated in a stack-based fashion according to the nesting of conditionals and loops. For program expression e we write $z(e)$ for the formal expression resulting from substituting each x in e with z^x. Similarly, for $F \in \mathcal{N}^*$, $F = [i_1, \ldots, i_n]$, we write $z(F)$ for the list $[z^{i_1}, \ldots, z^{i_n}]$.

A native command C is translated the into **join**-instrumented program $\iota(\epsilon, C)$ via the rules in Figure 3, where $F \in \mathcal{N}^*$ and $\mathbf{Join}(i, C)$ is given by

$$z^{x_1}{:=}\mathbf{join}\ \epsilon\ [z^i, z^{x_1}]; \ldots; z^{x_n}{:=}\mathbf{join}\ \epsilon\ [z^i, z^{x_n}]$$

C	$\iota(F,C)$	where ...
skip	**skip**	
$x{:=}e$	$x{:=}e; z^x{:=}\mathbf{join}\ \epsilon\ (z(e)@z(F))$	
$C_1; C_2$	$\iota(F,C_1); \iota(F,C_2)$	
if e **then** C_1 **else** C_2	$z^i{:=}\mathbf{join}\ [e]\ \alpha;$ **if** e **then** $\iota(i :: F, C_1); \mathbf{Join}(i, C_2)$ **else** $\iota(i :: F, C_2); \mathbf{Join}(i, C_1)$	$i \notin F$ $\alpha = z(F)@z(e)$
while e **do** C'	$z^i{:=}\mathbf{join}\ [e]\ \alpha_1;$ **while** e **do** $\iota(i :: F, C'); \mathbf{Join}(i, C');;$ $z^i{:=}\mathbf{join}\ [e]\ \alpha_2$ $\mathbf{Join}(i, C')$	$i \notin F$ $\alpha_1 = z(F)@z(e)$ $\alpha_2 = z^i :: \alpha_1$

Fig. 3. Synthesis of **join**-instrumented programs $\iota(F, C)$

where $x_1, \ldots, x_n$ is an (arbitrary) enumeration of $\mathsf{MV}(C)$.

Mirroring the effect of the annotations λ in the previous section, the translation of $x{:=}e$ updates z^x with the lub of the security registers associated with the variables in e and the dynamic taints of the enclosing branches as modeled by F. Composite statements $C_1; C_2$ are translated compositionally. Discarding any additions that may have been applied to F in $\iota(F, C_1)$, code $\iota(F, C_2)$ may well reuse some register z^i already in use in $\iota(F, C_1)$ (but not in F): the synthesis ensures that the liveness ranges of such z^i do not overlap, hence the variables in effect are distinct. Indeed, the translations of conditionals and loops initialize newly allocated z^i in their first instruction, by combining the data taint of the branch condition with its control taint $z(e)$ and the surrounding control flow taint $z(F)$, refining the use of $\Gamma(\mathbf{pc})$ in Figure 2. Bodies of conditionals and loops are translated compositionally by pushing the register i onto F, and are extended with compensation code using **Join**. Optimizing the behavior slightly in comparison to Figure 2, compensation code in conditionals is only added for variables modified in the opposite branch (in principle, adding compensation code for $\mathsf{MV}(C_2)\backslash\mathsf{MV}(C_1)$ in C_1 suffices, and similarly for C_2). Additionally, a loop body updates the security register z^i, thus propagating the taints of loop-controlling variables to the next iteration. By including i in α_2, we ensure that z^i monotonically increases, i.e. that information is appropriately propagated to later iterations. Finally, we add compensation code in a loop epilogue, ensuring that no information leaks from loops that are never entered.

Some simple properties of $\iota(F, C)$ are as follows:

Lemma 5. *Let* $(s, G, M) \xrightarrow{\iota(F,C)}_{\mathsf{J}} (t, D, N)$. *Then (i)* $M(z^x) = N(z^x)$ *for all* $x \notin \mathsf{MV}(C)$, *(ii)* $M(z^i) = N(z^i)$ *for all* $i \in F$, *and (iii)* $N(z^i) \sqsubseteq N(z^x)$ *for* $x \in \mathsf{MV}(C)$ *and* $i \in F$.

Example 2. Revisiting Example 1, we see that synthesis $\iota(F, C)$ generates code

$$z^0{:=}\mathbf{join}\ [x]\ [z^x];$$
$$\mathbf{if}\ x\ \mathbf{then}\ y{:=}v; z^y{:=}\mathbf{join}\ \epsilon\ [z(v), z^0]\ \mathbf{else}\ z^y{:=}\mathbf{join}\ \epsilon\ [z^0, z^y]$$

for the second conditional, where z^0 is a fresh taint register.The first instruction sets z^0 to the lub of the (dynamic) data and control taints of the branch condition x. In the

positive branch, this taint is then propagated to the control taint of y (together with the control taint of v). The negative branch is equipped with compensation code, lifting z^y to at least z^0. The code generated for the first conditional amounts to

$$z^0{:=}\mathbf{join}\ [e]\ (z(e));$$
$$\mathbf{if}\ e\ \mathbf{then}\ x{:=}m; z^x{:=}\mathbf{join}\ \epsilon\ [z(m), z^0]\ \mathbf{else}\ x{:=}h; z^x{:=}\mathbf{join}\ \epsilon\ [z(h), z^0]$$

where we have silently eliminated the compensation code $z^x{:=}\mathbf{join}\ \epsilon\ [z^0, z^x]$ that is formally appended to both branches, based on the observation that compensation code is redundant for variables modified in both branches. In particular, a run starting in (s, G, M) with $[\![e]\!]_s = \mathbf{true}$ only lifts z^x to the data and control taints of e and m, and hence correctly guarantees final-state-indistinguishability w.r.t. variable y for any execution starting in some state s' with $s =^{\Gamma_0}_{\sqsubseteq \kappa \sqcup mid} s'$.

The interpretation of join-instrumented executions combines the taint components for direct and indirect flows. We write $G \vartriangle M$ for the map that sends each program variable x to $Gx \sqcup M(z^x)$. We have proven results similar to those in Section 3; details are available in our Coq development [9]. In particular, the following result shows the agreement between an instrumented and a native execution, in the style of Corollary 1.

Theorem 4. *Let* $(s, G, M) \xrightarrow{\iota(F,C)}_{\mathsf{J}} (t, D, N)$ *and* $s' \xrightarrow{C} t'$. *For any* x, $s =^{G \vartriangle M}_{\sqsubseteq (D \vartriangle N)x} s'$ *implies* $tx = t'x$.

The proof proceeds by induction on C, again with case distinctions on $x \in \mathsf{MV}(C)$ in the cases for conditionals and loops, and an induction on the (instrumented) operational judgement in the case for loops where $x \in \mathsf{MV}(C)$.

Furthermore, the execution of $\iota(F, C)$ respects the static typing:

Lemma 6. *For* $\vdash \{\Gamma\}C\{\Delta\}$ *and* $(s, G, M) \xrightarrow{\iota(F,C)}_{\mathsf{J}} (t, D, N)$ *let* $\sqcup_{i \in F} M(z^i) \sqsubseteq \Gamma(\mathbf{pc})$ *and* $G \vartriangle M \sqsubseteq |\Gamma|$. *Then* $D \vartriangle N \sqsubseteq |\Delta|$.

In fact, $\iota(F, C)$ is more precise than $\theta(\mathcal{D}, C)$:

Theorem 5. *For* $\mathcal{D} \vdash \{\Gamma\}C\{\Delta\}$, $\sqcup_{i \in F} M(z^i) \sqsubseteq \Gamma(\mathbf{pc})$, *and* $G \vartriangle M \sqsubseteq |\Gamma|$ *let* $(s, G, M) \xrightarrow{\iota(F,C)}_{\mathsf{J}} (t, D, N)$ *and* $(s', G \vartriangle M) \xrightarrow{\theta(\mathcal{D},C)}_{\mathsf{T}} (t', D')$. *Then each* x *with* $s =^{G \vartriangle M}_{\sqsubseteq (D \vartriangle N)x} s'$ *satisfies* $tx = t'x \wedge (D \vartriangle N)x \sqsubseteq D'x$.

Example 2 is a typical case where $\iota(F, C)$ is *strictly* more precise than $\theta(\mathcal{D}, C)$.

5 Path Tracking

The previous sections focused on termination-insensitive security, a notion that is trivially satisfied whenever either execution fails to terminate. We now extend the synthesis so that termination is instead *propagated* from lead to minor executions.

The extension rests on the observation made in Section 3.3 that the execution paths taken by minor executions are to a large extent determined by the major execution, via

the relationships between the taints of dynamically encountered conditionals and the final taints of possibly-assigned variables. The exception to this rule are cases where the final value of a variable x in a major execution is independent from all assignments in a conditional, as in $(\mathbf{if}\ m\ \mathbf{then}\ y{:=}2\ \mathbf{else}\ y{:=}3); x{:=}5$, or indeed

$$(\mathbf{if}\ m\ \mathbf{then}\ y{:=}2\ \mathbf{else}\ \mathbf{while}\ \mathbf{true}\ \mathbf{do}\ \mathbf{skip}); x{:=}5.$$

In both cases, the final *low* taint of x does not constrain the value of $m : mid$ in a competitor initial state s'. Hence, competitor executions may follow different program paths. The same is true for cases where $x \notin \mathsf{MV}(C)$.

We modify synthesis $\iota(F, C)$ by adding a further security register, $z^{\mathbf{pc}}$ that collects the taints of *all* control-flow affecting expressions encountered during a (lead) run, in effect *tracking the (decisions determining the choice of) execution path*. Figure 4 presents the resulting synthesis $\xi(F, C)$. At each branch point, the taint held in $z^{\mathbf{pc}}$ is incremented by the direct and indirect taints of the control-flow affecting expression.

C	$\xi(F, C)$	where ...
skip	**skip**	
$x{:=}e$	$x{:=}e; z^x{:=}\mathbf{join}\ \epsilon\ (z(e)@z(F))$	
$C_1; C_2$	$\xi(F, C_1); \xi(F, C_2)$	
$\mathbf{if}\ e\ \mathbf{then}\ C_1$ $\mathbf{else}\ C_2$	$\boxed{z^{\mathbf{pc}}{:=}\mathbf{join}\ [e]\ \alpha_t}; z^i{:=}\mathbf{join}\ [e]\ \alpha;$ $\mathbf{if}\ e\ \mathbf{then}\ \xi(i :: F, C_1); \mathbf{Join}(i, C_2)$ $\mathbf{else}\ \xi(i :: F, C_2); \mathbf{Join}(i, C_1)$	$i \notin F$ $\alpha = z(F)@z(e)$ $\boxed{\alpha_t = z^{\mathbf{pc}} :: z(e)}$
$\mathbf{while}\ e\ \mathbf{do}\ C'$	$\boxed{z^{\mathbf{pc}}{:=}\mathbf{join}\ [e]\ \alpha_t}; z^i{:=}\mathbf{join}\ [e]\ \alpha_1;$ $\mathbf{while}\ e\ \mathbf{do}\ \xi(i :: F, C'); \mathbf{Join}(i, C');$ $\boxed{z^{\mathbf{pc}}{:=}\mathbf{join}\ [e]\ \alpha_t}; z^i{:=}\mathbf{join}\ [e]\ \alpha_2;$ $\mathbf{Join}(i, C')$	$i \notin F$ $\alpha_1 = z(F)@z(e)$ $\alpha_2 = z^i :: \alpha_1$ $\boxed{\alpha_t = z^{\mathbf{pc}} :: z(e)}$

Fig. 4. Path-tracking synthesis $\xi(F, C)$. Differences to Figure 3 are marked.

The following result combines the termination assurance with a claim similar to Theorem 4. Note that s and s' are still compared below $(D \triangle N)x$ rather than below the weaker $N(z^{\mathbf{pc}}) \sqcup (D \triangle N)x$. Indeed, $(D \triangle N)x \sqsubsetneq N(z^{\mathbf{pc}})$ typically holds whenever the most secret branch is encountered after the last assignment to x.

Theorem 6. *Let* $(s, G, M) \xrightarrow{\xi(F,C)}\!\!\!\!\!\rfloor\ (t, D, N)$ *and* $s =^{G \triangle M}_{\sqsubseteq N(z^{\mathbf{pc}})} s'$. *Then, there is some* t' *with* $s' \xrightarrow{C} t'$, *and we have* $tx = t'x$ *for any* x *with* $s =^{G \triangle M}_{\sqsubseteq (D \triangle N)x} s'$.

In the examples above, the lead executions for $s \vdash m \Downarrow \mathbf{true}$ yield $N(z^{\mathbf{pc}}) = (G \triangle M)m \sqcup M(z^{\mathbf{pc}})$. Thus, minor executions starting in states s' with $s =^{G \triangle M}_{\sqsubseteq N(z^{\mathbf{pc}})} s'$ necessarily satisfy $s'(m) = s(m)$, follow the same execution paths and hence terminate.

Synthesis $\xi(F, C)$ agrees with $\iota(F, C)$ on all taints other than $z^{\mathbf{pc}}$: writing $M \approx M'$ if $Mz = M'z$ for all $z \neq z^{\mathbf{pc}}$, we have that for $(s, G, M) \xrightarrow{\xi(F,C)}\!\!\!\!\!\rfloor\ (t, D, N)$ and

$$\text{TS-Ass}\frac{\Delta = \Gamma[x \mapsto \Gamma(\mathbf{pc}) \sqcup [\![e]\!]_\Gamma]}{\vdash_q \{\Gamma\}x{:=}e\{\Delta\}} \qquad \text{TS-Comp}\frac{\forall i \in \{1,2\}.\ \vdash_q \{\Gamma_i\}C_i\{\Gamma_{i+1}\}}{\vdash_q \{\Gamma_1\}C_1;C_2\{\Gamma_3\}}$$

$$\text{TS-Skip}\frac{}{\vdash_q \{\Gamma\}\mathbf{skip}\{\Gamma\}} \qquad \text{TS-Ite}\frac{\begin{array}{c} p = \Gamma(\mathbf{pc}) \qquad \Gamma' = \Gamma[\mathbf{pc} \mapsto p \sqcup [\![e]\!]_\Gamma] \\ \forall i \in \{1,2\}.\ \vdash_q \{\Gamma'\}C_i\{\Delta_i\} \qquad \boxed{p \sqcup [\![e]\!]_\Gamma \sqsubseteq q} \end{array}}{\vdash_q \{\Gamma\}\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\{(\Delta_1 \sqcup \Delta_2)[\mathbf{pc} \mapsto p]\}}$$

$$\text{TS-While}\frac{\begin{array}{c} \forall\, i < n.\ \Gamma_i \neq \Gamma_{i+1} \qquad \forall\, i \geqslant n.\ \Gamma_i = \Gamma_{i+1} \qquad p = \Gamma(\mathbf{pc}) \\ \Gamma_0 = \Gamma[\mathbf{pc} \mapsto p \sqcup [\![e]\!]_\Gamma] \qquad \forall i.\ \vdash_q \{\Gamma_i\}C\{\Delta_i\} \qquad \boxed{\Gamma_n(\mathbf{pc}) \sqsubseteq q} \\ \forall i.\ \Gamma_{i+1} = (\Delta_i \sqcup \Gamma)[\mathbf{pc} \mapsto (\Delta_i \sqcup \Gamma)(\mathbf{pc}) \sqcup [\![e]\!]_{\Delta_i \sqcup \Gamma}] \end{array}}{\vdash_q \{\Gamma\}\mathbf{while}\ e\ \mathbf{do}\ C\{\Gamma_n[\mathbf{pc} \mapsto p]\}}$$

Fig. 5. System $\vdash_q \{\Gamma\}C\{\Delta\}$. Differences to system $\vdash \{\Gamma\}C\{\Delta\}$ are marked.

C	$\delta(C)$
skip	**skip**
$x{:=}e$	$x{:=}e$
$C_1;C_2$	$\delta(C_1);\delta(C_2)$
if e **then** C_1 **else** C_2	$z^{\mathbf{pc}}{:=}\mathbf{join}\ [e]\ [z^{\mathbf{pc}}]$; **if** e **then** $\delta(C_1)$ **else** $\delta(C_2)$
while e **do** C'	$z^{\mathbf{pc}}{:=}\mathbf{join}\ [e]\ [z^{\mathbf{pc}}]$; **while** e **do** $(\delta(C'); z^{\mathbf{pc}}{:=}\mathbf{join}\ [e]\ [z^{\mathbf{pc}}])$

Fig. 6. Synthesis of termination-sensitive data-tracking programs $\delta(C)$

$(s,G,M') \xrightarrow{\iota(F,C)}_{\mathsf{J}} (t',D',N')$, $M \approx M'$ implies $t = t'$, $D = D'$ and $N \approx N'$. In particular, the claim in Theorem 5 remains valid if we replace $\iota(F,C)$ by $\xi(F,C)$.

The static counterpart to path tracking is an extension of the type system from Figure 1 to a system with judgements $\vdash_q \{\Gamma\}C\{\Delta\}$, where $q \in \mathcal{L}$ represents a (static) upper bound on the taints of branch conditions. We give the rules of this system in Figure 5 and note that $\vdash_q \{\Gamma\}C\{\Delta\}$ implies $\vdash \{\Gamma\}C\{\Delta\}$ and also $\vdash_p \{\Gamma\}C\{\Delta\}$ for any $p \sqsupseteq q$. A similar type system is given by Hunt and Sands [17], but not linked to dynamic taint tracking as we do in Theorem 7 below.

The property guaranteed by the static system is *equitermination* between executions starting in initial states that are Γ-indistinguishable below q:

Lemma 7. *Let* $\vdash_q \{\Gamma\}C\{\Delta\}$ *and* $s =^{|\Gamma|}_{\sqsubseteq q} s'$. *Then* $\exists t.\ s \xrightarrow{C} t$ *iff* $\exists\, t'.\ s' \xrightarrow{C} t'$.

Note that equitermination is symmetric and taint ignorant, i.e. applies to pairs of native executions (but can be extended to tainted executions by the erasure lemma).

Path tracking refines the static termination analysis, extending Lemma 6:

Theorem 7. *For* $\sqcup_{i\in F} M(z^i) \sqsubseteq \Gamma(\mathbf{pc})$ *let* $\vdash_q \{\Gamma\}C\{\Delta\}$ *and* $(s,G,M) \xrightarrow{\xi(F,C)}_{\mathsf{J}} (t,D,N)$. *If* $G \vartriangle M \sqsubseteq |\Gamma|$ *and* $M(z^{\mathbf{pc}}) \sqsubseteq q$ *then* $D \vartriangle N \sqsubseteq |\Delta|$ *and* $N(z^{\mathbf{pc}}) \sqsubseteq q$.

Interestingly, path tracking can also be carried out in the absence of control flow taints, i.e for pure data flow tracking. To this end, define the synthesis $\delta(C)$ by erasing from

$\xi(F, C)$ all **join** instructions (including those in **Join**) that define taint registers other than $z^{\mathbf{pc}}$, and modify instructions $z^{\mathbf{pc}}$:=**join** $[e]$ α to $z^{\mathbf{pc}}$:=**join** $[e]$ $[z^{\mathbf{pc}}]$ (see details in Fig. 6). The resulting programs ignore all control taint registers other than $z^{\mathbf{pc}}$, but behave exactly like $\xi(F, C)$ on data taints and the data plane: for $(s, G, M) \xrightarrow{\delta(C)}_{\mathsf{J}} (t, D, N)$, we have $M \approx N$, and $(t, D) = (t', D')$ whenever $(s, G, M') \xrightarrow{\xi(F,C)}_{\mathsf{J}} (t', D', N')$. Writing $\lfloor p \rfloor$ for the control taint component that sends $z^{\mathbf{pc}}$ to p and all other z to $\bot$, we also have that $\delta(C)$ stays below $\xi(F, C)$:

Theorem 8. *Let* $(s, G, \lfloor p \rfloor) \xrightarrow{\delta(C)}_{\mathsf{J}} (t, D, N)$ *and* $(s, G, M') \xrightarrow{\xi(F,C)}_{\mathsf{J}} (t', D', N')$. *Then,* $N = \lfloor q \rfloor$ *for some* $q \sqsupseteq p$, *with* $N \sqsubseteq N'$ *whenever* $p \sqsubseteq M'(z^{\mathbf{pc}})$. *In particular,* $u =^{D}_{\sqsubseteq\kappa} u'$ *coincides with* $u =^{D \triangle N}_{\sqsubseteq\kappa} u'$ *and implies* $u =^{D' \triangle N'}_{\sqsubseteq\kappa} u'$, *for all* u, u', *and* κ.

Even for arbitrary M, $\delta(C)$ enjoys termination and indistinguishability:

Theorem 9. *Let* $(s, G, M) \xrightarrow{\delta(C)}_{\mathsf{J}} (t, D, N)$ *and* $s =^{G}_{\sqsubseteq N(z^{\mathbf{pc}})} s'$. *Then, for any* G' *and* M', *there are* t', D', *and* N' *such that* $(s', G', M') \xrightarrow{\delta(C)}_{\mathsf{J}} (t', D', N')$. *Furthermore,* $tx = t'x$ *holds for any* x *with* $s =^{G}_{\sqsubseteq Dx} s'$, *and* $Dx = D'x$ *holds additionally whenever all* y *with* $Gy \sqsubseteq Dx$ *satisfy* $Gy = G'y$.

Finally, we transfer Theorem 9's conclusion to native executions and express security as an implication over multilevel indistinguishabilities using Lemma 3.

Corollary 2. *Let* $(s, G, M) \xrightarrow{\delta(C)}_{\mathsf{J}} (t, D, N)$ *and* $s =^{G}_{\sqsubseteq N(z^{\mathbf{pc}})} s'$. *Then,* $s' \xrightarrow{C} t'$ *for some* t', *and for all* κ, $s =^{G}_{\sqsubseteq\kappa} s'$ *implies* $t =^{D}_{\sqsubseteq\kappa} t'$.

To our knowledge, this represents the first extensional interpretation of data tracking.

6 Discussion

We presented an analysis of hybrid information flow by proving selected instrumentation schemes sound with respect to RIFLE-inspired interpretations of taint tracking.

Building upon Moore & Chong's analysis [24], we envision that our analysis can be extended to memory operations if side effects are tracked at the level of memory abstractions, for example by introducing one taint register (and associated compensation code) per region. Additional future work includes the support of (infinite) computations with output, a more detailed study of path tracking, and a comparison of taint tracking with Boudol's formulation of security as safety property [11,12].

Jee et al. [18] propose *taint flow algebras* as a generic framework for transferring traditional compiler optimizations to byte-level taint tracking. Fine-grained tracking below the level of words does not appear to have been studied in the language-based security community yet, but a unifying treatment of taint- and native optimizations may potentially emerge in explicitly relational formulations [8], extending Moore & Chong's use of two-level noninterference for selective monitoring.

Nanevski et al. [25] employ dependent types and relational Hoare Type Theory to enforce information flow and access control policies, although program verification is mostly carried out manually, by interactive verification in Coq.

Finally, separating control and data taints from each other and from the data plane appears in principle compatible with multicore execution, if the respective instruction streams are mapped onto different cores: as communication is orchestrated in an acyclic fashion, efficient loop pipelining may be enabled [26].

References

1. Amtoft, T., Dodds, J., Zhang, Z., Appel, A., Beringer, L., Hatcliff, J., Ou, X., Cousino, A.: A Certificate Infrastructure for Machine-Checked Proofs of Conditional Information Flow. In: Degano, P., Guttman, J.D. (eds.) POST 2012. LNCS, vol. 7215, pp. 369–389. Springer, Heidelberg (2012)
2. Appel, A.W.: Verified software toolchain - (invited talk). In: Barthe (ed.) [6], pp. 1–17
3. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Chong, S., Naumann, D. (eds.) PLAS 2009: Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pp. 113–124. ACM (2009)
4. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Banerjee, A., Garg, D. (eds.) PLAS 2010: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pp. 3:1–3:12. ACM (2010)
5. Austin, T.H., Flanagan, C., Abadi, M.: A functional view of imperative information flow. Technical Report UCSC-SOE-12-15, Department of Computer Science, University of California at Santa Cruz (2012)
6. Barthe, G. (ed.): ESOP 2011. LNCS, vol. 6602. Springer, Heidelberg (2011)
7. Barthe, G., Pichardie, D., Rezk, T.: A Certified Lightweight Non-interference Java Bytecode Verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
8. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM Symposium on Principles of Programming Languages, POPL 2004, pp. 14–25. ACM (2004)
9. Beringer, L.: End-to-end multilevel hybrid information flow control - Coq development (2012), `http://www.cs.princeton.edu/~eberinge/HybridIFC.tar.gz`
10. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: Proceedings of the 20th IEEE Computer Security Foundations Symposium, CSF 2007, pp. 233–248. IEEE Computer Society (2007)
11. Boudol, G.: On Typing Information Flow. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 366–380. Springer, Heidelberg (2005)
12. Boudol, G.: Secure Information Flow as a Safety Property. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 20–34. Springer, Heidelberg (2009)
13. Chudnov, A., Naumann, D.A.: Information flow monitor inlining. In: CSF 2010 [15], pp. 200–214 (2010)
14. Clause, J.A., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Rosenblum, D.S., Elbaum, S.G. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, pp. 196–206. ACM (2007)
15. Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010. IEEE Computer Society (2010)
16. Hunt, S., Sands, D.: On flow-sensitive security types. In: Morrisett, J.G., Jones, S.L.P. (eds.) Proceedings of the 33rd ACM Symposium on Principles of Programming Languages, POPL 2006, pp. 79–90. ACM (2006)
17. Hunt, S., Sands, D.: From exponential to polynomial-time security typing via principal types. In: Barthe (ed.) [6], pp. 297–316

18. Jee, K., Portokalidis, G., Kemerlis, V.P., Ghosh, S., August, D.I., Keromytis, A.D.: A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In: Proceedings of the 19th Network and Distributed System Security Symposium, NDSS 2012. The Internet Society, ISOC (2012)
19. Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proceedings of the 18th Network and Distributed System Security Symposium, NDSS 2011. The Internet Society, ISOC (2011)
20. Le Guernic, G.: Precise Dynamic Verification of Confidentiality. In: Beckert, B., Klein, G. (eds.) Proceedings of the 5th International Verification Workshop. CEUR Workshop Proceedings, vol. 372, pp. 82–96. CEUR-WS.org (2008)
21. Le Guernic, G., Jensen, T.: Monitoring Information Flow. In: Sabelfeld, A. (ed.) Proceedings of the Workshop on Foundations of Computer Security, FCS 2005, pp. 19–30. DePaul University (June 2005) (Affiliated with LICS 2005)
22. Magazinius, J., Russo, A., Sabelfeld, A.: On-the-fly Inlining of Dynamic Security Monitors. In: Rannenberg, K., Varadharajan, V., Weber, C. (eds.) SEC 2010. IFIP AICT, vol. 330, pp. 173–186. Springer, Heidelberg (2010)
23. Masri, W., Podgurski, A., Leon, D.: Detecting and debugging insecure information flows. In: Proceedings of the 15th International Symposium on Software Reliability Engineering, ISSRE 2004, pp. 198–209. IEEE Computer Society (2004)
24. Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, pp. 146–160. IEEE Computer Society (2011)
25. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: 32nd IEEE Symposium on Security and Privacy, S&P 2011, pp. 165–179. IEEE Computer Society (2011)
26. Rangan, R., Vachharajani, N., Vachharajani, M., August, D.I.: Decoupled software pipelining with the synchronization array. In: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT 2004, pp. 177–188. IEEE Computer Society (2004)
27. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: CSF 2010 [15], pp. 186–199 (2010)
28. Sabelfeld, A., Russo, A.: From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) PSI 2009. LNCS, vol. 5947, pp. 352–365. Springer, Heidelberg (2010)
29. Vachharajani, N., Bridges, M.J., Chang, J., Rangan, R., Ottoni, G., Blome, J.A., Reis, G.A., Vachharajani, M., August, D.I.: Rifle: An architectural framework for user-centric information-flow security. In: 37th Annual International Symposium on Microarchitecture (MICRO-37), pp. 243–254. IEEE Computer Society (2004)
30. Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably Correct Runtime Enforcement of Non-interference Properties. In: Ning, P., Qing, S., Li, N. (eds.) ICICS 2006. LNCS, vol. 4307, pp. 332–351. Springer, Heidelberg (2006)