Optimising Faceted Secure Multi-Execution

Maximilian Algehed
Computer Science and Engineering
Chalmers
Gothenburg, Sweden
algehed@chalmers.se

Alejandro Russo
Computer Science and Engineering
Chalmers
Gothenburg, Sweden
russo@chalmers.se

Cormac Flanagan
Computer Science and Engineering
University of California, Santa Cruz
Santa Cruz, USA
cormac@ucsc.edu

***Abstract*—Language-Based Information Flow Control (IFC) provides strong security guarantees for untrusted code, but often suffers from a non-negligible rate of false alarms. Multi-execution based techniques promise to provide security guarantees without raising any false alarms. However, all known multi-execution approaches introduce extraneous performance overheads which are rarely studied. In this work, we lay down the foundations for optimisation techniques aimed at reducing these overheads to a managable level, thus helping to make multi-execution *more practical*. We characterise our optimisations as data- and control-oriented. Data-oriented optimisations reduce storage overheads— which also helps to remove unnecessary repeated computations. In contrast, computation-oriented optimisations rely on program annotations in order to reduce needless computation. These annotations motivate the need for a new, stronger, theoretical notion of transparency— i.e., a stronger notion for characterising the lack of false alarms. To show the efficacy of our optimisation techniques, we apply them to two case-studies: a secure (faceted) database and a chat server written in a multi-execution based IFC framework. Our case-studies clearly show that our optimisations significantly reduce the storage and computational overhead, sometimes from exponential to polynomial order. All of our formal results are accompanied by mechanised proofs in Agda.**

***Keywords*-Faceted Values, Secure Multi-Execution, Multiple-Facets, Optimisation, Information Flow Control**

I. Introduction

Information flow control [1] (IFC) is a promising technology for preserving confidentiality of data. Many IFC approaches are designed to prevent sensitive data from influencing what attackers can observe from a program's public behavior—a security policy known as non-interference [2]. Researchers have proposed numerous enforcement mechanisms for non-interference based on, for instance, type systems [3, 4, 5] as well as execution monitors [6, 7]. While sound w.r.t. non-interference, such IFC approaches typically suffer from false alarms [8, 9, 10]. This phenomenon occurs due to IFC mechanisms sometimes needing to be overly cautious.

To remove false alarms, researchers have recently proposed IFC techniques based on multi-execution [9, 11, 12, 13]: *many copies of a given program (or parts of it) get executed while carefully adapting their semantics to avoid information leakage*. Secure Multi-Execution [11] (SME) and Multiple Facets [9] (MF) are two approaches based on this idea. The price to pay for multi-execution, however, is a degradation in both *computation time* and *memory consumption* due to repeated computations.

Faceted Secure Multi-Execution [12] (FSME) is a recent multi-execution technique capable of adjusting the performance of multi-executions by exchanging degradation of memory consumption for computation time and vice versa. FSME achieves both of these while ensuring non-interference in a setting involving a decentralised security lattice [14], a lattice which allows mutually distrusting principals to independently impose confidentiality and integrity requirements on data. By adjusting the trade-off, FSME can behave as SME, MF, or something "in between"—thus providing a unifying framework for multi-execution. This unification has enabled the first apples-to-apples comparison and benchmarking of the mentioned multi-execution techniques—which before were only contrasted in either an informal qualitative [9] or theoretical [15] manner.

FSME is attractive for building systems since developers can choose where performance degradation can be tolerated by offering a choice between memory consumption and computation time. In this work, we argue that this trade-off is not enough to build practical systems: *memory and time can be exponential in the number of principals aggregating data*. In these (not uncommon) situations, FSME programs could trade-off a large amount of memory for a long computing time or vice verse—an undesirable situation regardless what resource developers favor. This exponential blowup is not exclusive of FSME and also occurs in MF and SME when considering decentralised or large lattices [13]. Furthermore, the memory blowup can be translated to persistent storage when serialising data into databases [16]. In this light, it becomes necessary to introduce optimisations that mitigate both time and memory consumption.

In this work, we present novel optimisations for the FSME framework which we classify as *data-* and *computation-*oriented, respectively. The former are designed to reduce memory consumption as well as unnecessary repeated computations, while the latter aim to reduce computation time alone. We demonstrate, both formally and empirically, that our optimisations can change an exponential consumption of resources to more manageable polynomial sized overheads. Our techniques work for a wide-range of security lattices, from finite to decentralised ones. We also note that while our empirical contributions concern the Multef framework, which implements FSME, the insights of this work also apply in other multi-execution settings. Furthermore, all our claims have been

mechanised in the Agda proof assistant and are available as supplimentary material to this paper [1].

Data-oriented optimisations are designed to simplify the internal representation of values in FSME as well as faceted databases [16]. Values are represented by a tree-structure, where each leaf captures a view for a given set of observers. For instance, the faceted value $\langle \text{Alice} \ ? \ 42 : 0 \rangle$ indicates that the (secret) value 42 belongs to the principal Alice, while any other principal will see the corresponding public value 0. Programs operate in a manner which is agnostic to the faceted structure. All operations in the language work uniformly on faceted values in a way which respects their structure. Such operations often result in deeply nested trees. For instance, $\langle \text{Alice} \ ? \ 42 : 0 \rangle + \langle \text{Bob} \ ? \ 5 : 2 \rangle$ results in $\langle \text{Alice} \ ? \ \langle \text{Bob} \ ? \ 47 : 44 \rangle : \langle \text{Bob} \ ? \ 5 : 2 \rangle \rangle$ — a tree of depth two and with four leaves. It is easy to imagine how quickly faceted values can grow if nothing is done to prevent it.

In this work, we present a novel set of rewrite rules capable of removing redundant leaves in faceted values. For that, we first formalise an equivalence relation to ensure that our rules do not change the observable views (i.e., who observes what) encoded by faceted values (see Section III). In a nutshell, the rewrite rules are targeted at shrinking nested structures based on the relation among parent- and child-labels. However, to maximise rewriting opportunities, we build on the notion of lattice residuation [17] in order to *systematically simplify children's labels based on the parents'*—a step which may lead to further simplifications. While residuation works out of the box for standard security lattices, it is more challenging for decentralised ones. We provide, to the best of our knowledge, the first definition of residuation for the decentralised security lattice of DC-labels [14]. We also present a case study (based on [16]) where our rewriting rules shrink faceted values until they become manageable in size.

Computation-oriented optimisations, on the other hand, aim to *avoid introducing leaves* that are never meant to be accessed by the computation. For instance, if we know that a sub-computation will only ever write its result to Bob's file, it makes sense to never compute the values of leaves supposed to be seen by Alice. More generally speaking, if we know that a computation writes to a sink at level ℓ, it makes sense to only focus on computing leaves for observers at level ℓ' such that $\ell' \sqsubseteq \ell$—the other leaves will never be accessed by the computation and thus they are not needed! While it is simple to see how this works when a computation is known to write only to a single level, we show that the idea extends to more complex situations as well. In previous work [9, 12, 13], authors provide a notion of *projection* responsible for obtaining from a faceted value the value observable at a certain level, written $f \downarrow \ell$ (where f is a faceted value)—observe that this operator is almost exactly what we need! Unfortunately, the projection operator has never been considered as part of the language, but as an artifact of the proof technique used for proving security and the absence of false alarms—a property known as *transparency*.

One contribution of this work is to show how to internalise the projection operator into the language in a sound way, written $f \Downarrow e$, where e is a boolean expression over labels (explained below). In this manner, developers can use it as a mechanism to explicitly avoid computing unnecessary leaves, thus saving computing time. By considering boolean expressions (see Section IV), developers can naturally specify which leaves of a given faceted value to compute. For example, $f \Downarrow \neg\text{Alice}$ will throw away the leaves corresponding to Alice while $f \Downarrow (\neg\text{Alice} \wedge \text{Bob})$ does the same but keeping the leaves at least as sensitive as Bob. Unfortunately, the internalisation of projection is incompatible with existing formulations of transparency [9, 12, 13] (Section IV). In this light, we introduce the notion of *focused transparency*, which says that transparency holds for those security levels which match the assumptions encoded in $\Downarrow$-projections. We show that this notion of transparency subsumes the traditional one and propose it as a new, stronger, alternative to the old definition. We formalise the pure part of FSME as a lambda calculus and show that programs executed by it fulfill both non-interference and focused transparency. The main difference between the calculus presented here and FSME [12] is that ours lacks side-effects, namely references and I/O. We argue that our optimisations naturally extend to these features (see Section IV).

To summarise, the contributions of this paper can be categorised as follows.

► Novel optimisations to make multi-execution based approaches, like [9, 12, 18, 19], more likely to be usable in practise.

► Data-oriented optimisations:

▷ A set of rewrite rules capable of shrinking a faceted value without changing its semantics.

▷ A novel notion of residuation for security lattices, which also covers decentralised label models like DC-labels [20].

▷ An Agda mechanisation showing that our that our rewrite rules do not affect the semantics of faceted values.

► Computation-oriented optimisations:

▷ Identification of the need for computation-oriented optimisations both from a theoretical and a practical perspective.

▷ Introduction of a language-level projection operator based on a boolean algebra to express what views to compute for.

▷ Introduction of the notion of *focused transparency*, a stronger notion than traditional transparency.

▷ Introduction of a pure core-calculus which models a essential functionality FSME.

▷ An Agda mechanisation showing that programs running in this calculus fulfill non-interference as well as focused transparency.

► Case studies that show how our optimisations help reduce the exponential resource consumption associated with multi-

[1] https://github.com/OctopiChalmers/OptimisingFSME

execution to a reasonable level.

II. Background

A. Security Lattices

A Security Lattice [21] encodes the allowed flows of information using a set $\mathscr{L}$ of security labels together with an order $\sqsubseteq$ (often pronounced "can flow to") and operations for combining security concerns, $\sqcup$ and $\sqcap$. Data labeled ℓ_L can flow to a data sink labeled ℓ_H if and only if $\ell_L \sqsubseteq \ell_H$. The join ($\sqcup$) and meet ($\sqcap$) operations compute least upper bounds and greatest lower bounds respectively. In this work, we consider lattices with a greatest element ($\top$) called "top", and a least ($\bot$) called "bottom."

There are a number of examples in the literature of useful security lattices. In this paper, we start by focusing on the *powerset of principals lattice*. The points in this lattice are drawn from the set of sets of principals, written $\mathscr{P}(\mathscr{A})$, where $\mathscr{A}$ is some (possibly non-finite) set of actors or principals. This lattice can express the concerns of multipl principals simulatenously. For example, data labeled with $\{\text{Alice}, \text{Bob}\}$ indicates that both principal have confidentiality concerns for the information. The order relation is given by subset, $\ell \sqsubseteq \ell' \iff \ell \subseteq \ell'$, with $\bot = \emptyset$, $\top = \mathscr{A}$, $\sqcup = \cup$, and $\sqcap = \cap$. For ease of reading we write singleton labels like $\{\text{Alice}\}$ without the curly brackets, simply as Alice.

Another interesting security lattice is Disjunction Category Labels or DC-labels for short [22]. A DC-label is a pair of labels, expressing confidentiality and integrity concerns respectively. More precisely, a label consists of a monotone logical formula with principals for atoms. For instance, when considering confidentiality, labels like Alice simply expresses that data is confidential to the principal Alice and Alice $\sqsubseteq \ell$ if and only if $\ell \Rightarrow \text{Alice}$, i.e., the label ℓ is strong enough to preserve Alice's concerns. In general, for any two confidentiality labels ℓ and ℓ' we have that $\ell \sqsubseteq \ell'$ if and only if $\ell' \Rightarrow \ell$. Conjunction expresses joint security concerns, data which is sensitive to *both* Alice and Bob is labeled $\text{Alice} \wedge \text{Bob}$. In this case, the flow $\text{Alice} \wedge \text{Bob} \sqsubseteq \text{Alice} \wedge \text{Bob} \wedge \text{Charlie}$ is allowed, but $\text{Alice} \wedge \text{Bob} \not\sqsubseteq \text{Alice}$. Similarly, the label $\text{Alice} \vee \text{Bob}$ expresses that either Alice or Bob can observe this data separately.

Integrity labels are dual to confidentiality labels. For any two integrity labels ℓ and ℓ' we have that $\ell \sqsubseteq \ell'$ if and only if $\ell \Rightarrow \ell'$. Putting a confidentiality and an integrity label together forms a DC-label, which we write as a pair (ℓ_c, ℓ_i), where ℓ_c is a confidentiality label while ℓ_i an integrity one. Naturally, we get that $(\ell_c, \ell_i) \sqsubseteq (\ell'_c, \ell'_i)$ if and only if $\ell'_c \Rightarrow \ell_c \wedge \ell_i \Rightarrow \ell'_i$. DC-labels form a decentralised lattice that has been used in various implementations of IFC systems [7, 12, 23, 24, 25]. DC-labels are also capable of modeling other decentralised models, such as DLM [26].

B. Faceted Values

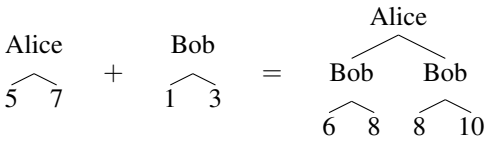
The set of faceted values over values v from V with labels ℓ from $\mathscr{L}$ is defined by the following grammar:

$$\text{Faceted}(V) \ni f \ ::= \ \langle \ell \ ? \ f : f \rangle \mid v$$

A faceted value captures different *views* on data based on the potential observers. For simplicity, we focus on confidentiality from now on. To illustrate how faceted values work, we consider the $\mathscr{P}(\mathscr{A})$ lattice described in Section II-A. In essence, the faceted value $\langle \text{Alice} \ ? \ 5 : 7 \rangle$ should behave as 5 to anyone who is allowed to see Alice's private data, and 7 to everyone else. Importantly, operations on faceted values respect the different *views of principals*. For instance, adding two faceted values like $\langle \text{Alice} \ ? \ 5 : 7 \rangle + \langle \text{Bob} \ ? \ 1 : 3 \rangle$ results in:

$$\langle \text{Alice} \ ? \ \langle \text{Bob} \ ? \ 6 : 8 \rangle : \langle \text{Bob} \ ? \ 8 : 10 \rangle \rangle$$

Note that if the observer of the addition sees both Alice's and Bob's data, then the result is 6, which comes from adding 5 in $\langle \text{Alice} \ ? \ 5 : 7 \rangle$ and 1 in $\langle \text{Bob} \ ? \ 1 : 3 \rangle$. However, if the observer is precisely Bob, the result is 8, which comes from adding 7 in $\langle \text{Alice} \ ? \ 5 : 7 \rangle$ and 1 in $\langle \text{Bob} \ ? \ 1 : 3 \rangle$. In this work, we use both the formal notation for faceted values as well as a graphical representation in the form of trees. For instance, the addition above can be described by the following trees:



The theory of faceted values comes equipped with a *projection* function responsible for extracting a value based on the security level of the observer. This function simply navigates through the tree structure until it finds the leaf that corresponds to the "right" value. Formally, projection is defined as a binary function $\downarrow$, such that if an observer at level ℓ_o observes t, she sees $t \downarrow \ell_o$:

$$\langle \ell \ ? \ f_1 : f_2 \rangle \downarrow \ell_o = \begin{cases} f_1 \downarrow \ell_o, & \ell \sqsubseteq \ell_o \\ f_2 \downarrow \ell_o, & \text{otherwise} \end{cases}$$

$$v \downarrow \ell_o = v$$

With the definition of projection in place, we can state properties about well-behaved faceted operations. For instance, the addition of faceted values should respect the different views found in faceted values:

$$\forall \ell_o. \ (t_1 + t_2) \downarrow \ell_o = (t_1 \downarrow \ell_o) + (t_2 \downarrow \ell_o)$$

Note that the addition on the left-hand side is on faceted values, while the one on the right is for numbers. The equation says that, *for all observers*, the resulting faceted values should be consistent with the addition. This idea of preserving behavior across all the observers motivates the following definition of equivalence for faceted values.

Definition 1 (Equivalence): We say that t_1 and t_2 are equivalent under projection, written $t_1 \sim t_2$, if and only if $t_1 \downarrow \ell = t_2 \downarrow \ell$ for all $\ell \in \mathscr{L}$.

The power of this equivalence relation is that programs which treat faceted values securely are guaranteed to preserve it, and therefore we are able to freely interchange equivalent values in computation. In Section III we show how careful study of this equivalence relation gives rise to a number of optimisation rules for faceted values.

C. Residuated Lattices and Galois Connections

In its most general form, a *residuated lattice* [17] is a lattice together with a monoid (it has an associative multiplication with an identity) which has a kind of "inverse" called the *residuation.* The relation between the multiplication and the residuation is captured by a Galois connection (explained below) [27]. For the purpose of this paper, we focus on two special cases of residuated lattices, where the multiplication operation is either the join ($\sqcup$), with residual $\ominus$, or meet ($\sqcap$), with residual $\oslash$, of the lattice.

Definition 2 (Join Residuated Lattice): We say that a lattice $\mathscr{L}$ is *join-residuated* if and only if there exists a binary operation $\ominus$ on $\mathscr{L}$ such that for all ℓ, $D_\ell(\ell') = \ell' \ominus \ell$ and $M_\ell(\ell') = \ell' \sqcup \ell$ form a Galois connection (written $D_\ell \dashv M_\ell$):

$$\forall \ell_1, \ell_2 . D_\ell(\ell_1) \sqsubseteq \ell_2 \iff \ell_1 \sqsubseteq M_\ell(\ell_2)$$

Intuitively, the equivalence above says that if ℓ_1 flows to the "multiplication" of ℓ_2 with ℓ ($\ell_1 \sqsubseteq M_\ell(\ell_2)$), then we can "divide" the inequality by ℓ, thus resulting in just ℓ_2 on the right-side, while the left-hand side is just ℓ_1 divided by ℓ ($D_\ell(\ell_1) \sqsubseteq \ell_2$).

Dually, we also consider residuation with respect to meets.

Definition 3 (Meet-residuated Lattice): We say that a lattice $\mathscr{L}$ is *meet-residuated* if and only if there exists a binary operation $\oslash$ on $\mathscr{L}$ such that for all ℓ, $M_\ell(\ell') = \ell' \sqcap \ell$ and $D_\ell(\ell') = \ell' \oslash \ell$ form a Galois connection (written $M_\ell \dashv D_\ell$):

$$\forall \ell_1, \ell_2 . M_\ell(\ell_1) \sqsubseteq \ell_2 \iff \ell_1 \sqsubseteq D_\ell(\ell_2)$$

We remark that is not always possible to take an arbitrary lattice and make it join- or meet-residuated. However, many lattices commonly studied in IFC are join residuated. For example, the powerset set of principals lattice has for its join residuation the relative compliment, or "set minus" operation defined as $\ell \ominus \ell' = \{p \mid p \in \ell, p \notin \ell'\}$. It is not hard to see that the standard two-point lattice is also join residuated (by observing that it is isomorphic to $\mathscr{P}(\mathbb{1})$). However, it is not clear if residuation works with more complex lattices like DC-labels. In Section III-B, we show that DC-labels are both join and meet residuated. This contribution allows us to apply the optimisation techniques presented in the following sections to practical applications.

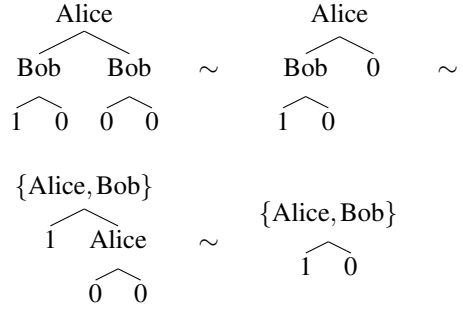
III. Data-Oriented Optimisations

In this section we present a number of useful and instructive semantics preserving equivalences for faceted values—as defined in Definition 1. These equivalences are designed to remove as much redundant information as possible from the tree-structure found in faceted values. We begin by considering optimisations which work for faceted values where the labels are drawn from an arbitrary security lattice, and proceed to consider the (ubiquitous) special case of residuated lattices.

We split the equivalences in two kinds: those that remove substructures in faceted values, shown in Figure 1, and those that simply rewrite them by exchanging leaves or nodes, shown in Figures 2 and 4. These equivalences can be used as part of a rewriting system that applies, for instance, the rules in Figure 1 until no rules apply, and then switches to apply those in Figures 2 and 4 to obtain a faceted value where it is again possible to apply the rules in Figure 1, and so on.

The rules in Figure 1 are equivalences that reduce the size of faceted values. Rule Choice Irrelevance eliminates redundancy in the form of duplicated values. Rule Bottom Irrelevance is equivalent to saying that $\bot$ does not protect any information and gets rid this unnecessary label. Rules Left Squash and Right Squash remove redundancy by exploiting the lattice structure. Observe that the last two rules require a relationship among the left- (right-) child and the parent node.

The $\sim$ relation does not just allow us to remove unnecessary labels from the faceted tree. We also present some equivalences which deal with rotations of faceted tree—see Figure 2. For example, Join uses least-upper-bounds to rotate a faceted tree from left to right. Similarly, Qualified Rotation allows rotation back and forth given that $\ell_2 \sqsubseteq \ell_1$. Rotations are crucial as they may expose further opportunities to remove duplicates using the Choice Irrelevance rule. For example, consider $\langle \text{Alice ? } 1 : 0 \rangle \times \langle \text{Bob ? } 1 : 0 \rangle$, which is equal to $\langle \text{Alice ? } \langle \text{Bob ? } 1 : 0 \rangle : \langle \text{Bob ? } 0 : 0 \rangle \rangle$, and can be shrunk considerably:



The derivation uses Choice Irrelevance, then Join, followed by Choice Irrelevance again. All in all, the algebraic structure of faceted values is rich and we encourage the interested reader to explore the theory at their leisure, and refer them to the Agda mechanisation for a number of other equivalences for rotations and similar equivalences with their equivalence proofs.

Next we focus our attention on other equivalences dedicated to the labels in the tree. By doing so, it may become possible to subsequently apply other equivalences like those in Figure 1. To illustrate this point, consider the tree in Figure 3 with labels drawn from the powerset lattice: We argue that this faceted value is equivalent to the much simpler $\langle \text{Bob ? } 1 : 2 \rangle$. However, this is not easily derived from the equivalences presented above. We could consider replacing $\{\text{Alice}, \text{Bob}\}$ by just Bob since Alice is already present as

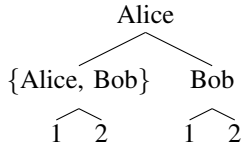


Fig. 3. A motivating example

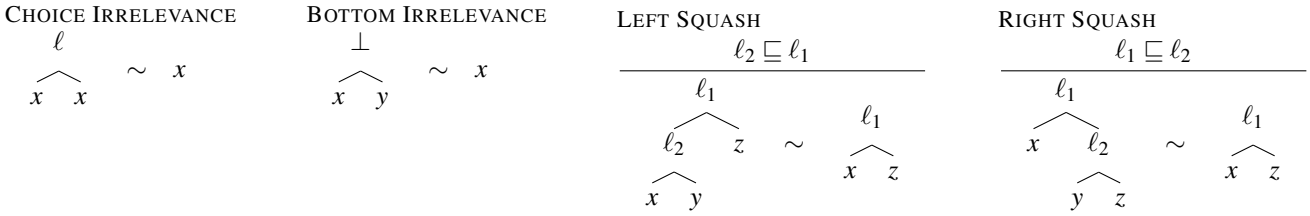


Fig. 1. Semantics preserving optimisations

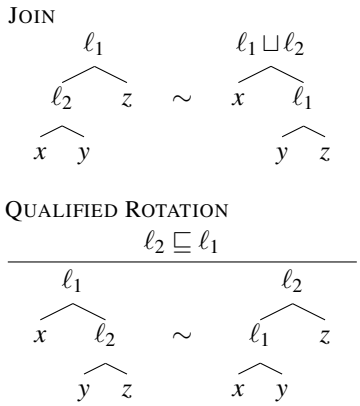
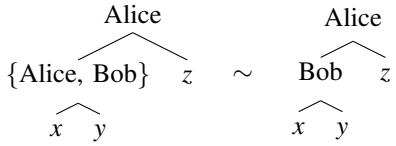
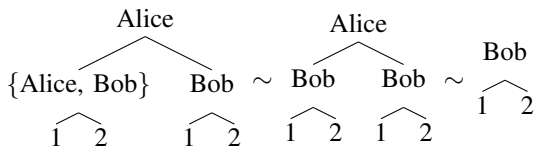


Fig. 2. Semantics preserving manipulations

root of the left-hand child and thus redundant. In other words, we hope that the following equivalence holds:



We need to make sure that replacing $\{\text{Alice}, \text{Bob}\}$ by Bob does not affect the observers of x and y. On the left-hand side of the equivalence, we have that the observers of x are those at level ℓ_o such that $\text{Alice} \sqsubseteq \ell_o$ (root) and $\{\text{Alice}, \text{Bob}\} \sqsubseteq \ell_o$ (left children). With these two facts, we can deduce that *it must be the case* that $\text{Bob} \sqsubseteq \ell_o$. Hence, observers of x on the left-hand side tree are also observers of x in the right-hand side tree. In a similar manner, we can prove that all the observers of x in the right-hand side tree are also observers in the left-hand one. This reasoning shows that the observers of x are the same on both trees. Similar reasoning can be applied to the observers of y to finally conclude that the two trees are equivalent. Note that we can now simplify labels first, and then use CHOICE IRRELEVANCE to remove some faceted values, which gives us the following derivation:



We face two challenges when generalising this approach to arbitrary faceted values: (a) *how do we know if a label can replace another one?* and (b) *how do we simplify an existing label as much as possible?* Motivated by challenge (a), we characterise the general notion of a label ℓ_2 replacing a label ℓ_1 to the left or the right of a parent label ℓ.

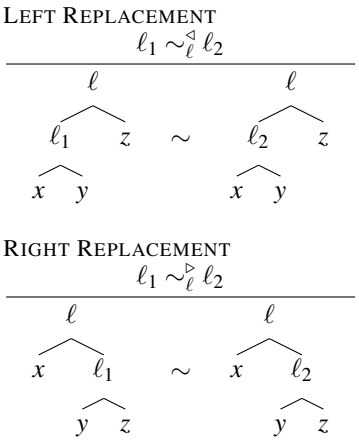


Fig. 4. Semantics preserving label replacement

Definition 4 (Replaces to the Left/Right): We say that a label ℓ_2 replaces ℓ_1 to the left (right) of ℓ, written $\ell_2 \sim_\ell^\triangleleft \ell_1$ (resp. $\ell_2 \sim_\ell^\triangleright \ell_1$), when for all ℓ_o we have that:

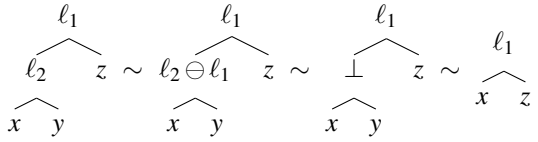
$$\ell \sqsubseteq \ell_o \Rightarrow (\ell_1 \sqsubseteq \ell_o \iff \ell_2 \sqsubseteq \ell_o)$$

$$(\text{resp. } \ell \not\sqsubseteq \ell_o \Rightarrow (\ell_1 \sqsubseteq \ell_o \iff \ell_2 \sqsubseteq \ell_o))$$

Note that $\sim_\ell^\triangleleft$ and $\sim_\ell^\triangleright$ are equivalence relations. This definition lets us derive the LEFT REPLACEMENT and RIGHT REPLACEMENT equivalences in Fig. 4. While this definition characterises when it is possible to replace a label ℓ_1 by another, ℓ_2, how do we know what the simplest label to use is? To address this challenge, (b), we turn to residuated lattices.

Theorem 1: For a join residuated lattice $\mathscr{L}$ and any labels $\ell_1, \ell_2 \in \mathscr{L}$, the residual $\ell_2 \ominus \ell_1$ is the least label which replaces ℓ_2 to the left of ℓ_1

When considering residuated lattices and the theorem above, it turns out that we can derive LEFT SQUASH rule from above. Assuming $\ell_2 \sqsubseteq \ell_1$ we have:



The second step, i.e., rewriting $\ell_2 \ominus \ell_1$ to $\bot$, is due to a theorem of join residuated lattices that says that $\ell_2 \sqsubseteq \ell_1 \Leftrightarrow \ell_2 \ominus \ell_1 = \bot$.

A. Constructing Residuated Lattices

For distributive lattices, i.e., where $\sqcup$ distributes over $\sqcap$, residuation follows mechanically. The following known theorem of residuated lattices make this point clear.

Theorem 2 (From [17]): If $\mathscr{L}$ is a lattice, then $\mathscr{L}$ is join-residuated given the following:

$$\forall \ell, L \subseteq \mathscr{L}.\ \ell \sqcup \bigsqcap L = \bigsqcap \{\ell \sqcup \ell' \mid \ell' \in L\}$$

Note that the condition says that least upper bounds distribute over (perhaps non-finite) meets. An immediate consequence of the definition of join-residuation tells us that $\ell \ominus \ell'$ is the *least* label such that $\ell \sqsubseteq (\ell \ominus \ell') \sqcup \ell'$. By applying the theorem above, and in order to obtain a join-residuated lattice, we simply construct $\ell \ominus \ell'$ as the meet over all labels ℓ_r such that $\ell \sqsubseteq \ell_r \sqcup \ell'$. Note that one simple consequence of this line of reasoning is that, for all *finite* lattices satisfying Theorem 2, we have a constructive (albeit potentially expensive) way of computing the residual. However, in the infinite cases like those of decentralised lattices, we do not get such an algorithm for free.

B. Residuation of DC-labels

In this section, we give (to the best of our knowledge) the first algorithm for computing the residuation for a decentralised (and thus infinite) security lattice. To illustrate how our construction works, we focus only on the confidentially part of DC-labels since integrity follows by duality. The confidentiality part of a DC-label is a monotone logical formula, which we express here in conjunctive normal form as sets of sets of principals. The set of CNFs over $\mathscr{A}$, denoted $\texttt{CNF}(\mathscr{A})$, is defined as:

$$\texttt{CNF}(\mathscr{A}) = \mathscr{P}(\mathscr{P}(\mathscr{A}))$$

Intuitively, elements of $\mathscr{P}(\mathscr{A})$ represent disjunctions of principals (called clauses) which are put together in a set $\mathscr{P}(\mathscr{P}(\mathscr{A}))$ representing their conjunction. We introduce some terminology.

Definition 5 (Discharged clauses): Given $c \in \mathscr{P}(\mathscr{A})$, we say that a clause c is *discharged* by a label ℓ, written $\ell \supset c$, if and only if $\exists c' \in \ell.\ c' \subseteq c$.

Using this definition, we define join residuation as follows.

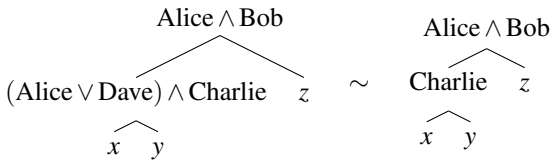
Definition 6 (Join-residuation for $\texttt{CNF}(\mathscr{A})$): Given $\ell, \ell' \in \texttt{CNF}(\mathscr{A})$, we define join-residuation as

$$\ell \ominus \ell' = \{\ c \in \ell \mid \ell' \not\supset c\ \}$$

Note the similarity to the definition of residuation for the powerset lattice, instead of removing elements of ℓ' from ℓ, we remove clauses implied by ℓ' from ℓ. We prove that this definition is sound.

Theorem 3: $\texttt{CNF}(\mathscr{A})$ is join-residuated.

With join residuation in place, we can see LEFT REPLACEMENT in action on faceted values which only consider confidentiality.



In the example above, $((\text{Alice} \vee \text{Dave}) \wedge \text{Charlie}) \ominus (\text{Alice} \wedge \text{Bob})$ results in Charlie—intuitively, the residuation removes the clause involving Alice since it appears in the root node.

Next we consider meet-residuation. Recall that the meet of two formulae ℓ and ℓ' is precisely their disjunction $\ell \vee \ell'$ (see Section II-A). When we frame this in terms of the sets of sets formulation we have been considering, we end up with the following definition of meet:

$$\ell \sqcap \ell' = \{\ c \cup c' \mid c \in \ell, c' \in \ell'\ \}$$

With this definition in mind, we define the meet-residuation of Definition 3 as follows.

Definition 7 (Meet-residuation for $\texttt{CNF}(\mathscr{A})$): Given $\ell, \ell' \in \texttt{CNF}(\mathscr{A})$, we define meet-residuation as

$$\ell \oslash \ell' = \bigsqcap \{\{c - c' \mid c \in \ell\} \mid c' \in \ell'\}$$

Theorem 4: $\texttt{CNF}(\mathscr{A})$ is meet-residuated.

By duality, the definitions in Theorems 4 and 3 respectively provides us a join- and meet-residuation also when formulas are seen as integrity requirements, which we write $\texttt{CNF}(\mathscr{A})^{\texttt{OP}}$. In turn this gives us join- and meet-residuation for DC-labels by simply considering the product lattice $\texttt{CNF}(\mathscr{A}) \times \texttt{CNF}(\mathscr{A})^{\texttt{OP}}$.

Corollary 1: The lattice induced by DC-labels is join- and meet-residuated.

C. Context-aware optimisations

A lot of the equivalences we have discussed are designed to exploit the information provided by the parent node (e.g., LEFT SQUASH, LEFT REPLACEMENT). For a motivating example of why this may sometimes be insufficient, consider the faceted value in Figure 5.

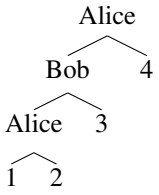


Fig. 5. A difficult to shrink tree

Optimising this tree should be a simple case of removing the redundant $\langle \text{Alice ? } 1 : 2 \rangle$ subtree and replacing it with the leaf 1, because Alice already appears above this subtree. However, none of the rules presented so far allow us to perform this simple optimisation, the LEFT SQUASHrule for example only takes into consideration the immediate parent-child relationship between nodes.

To remedy this shortcoming we introduce new optimisations by generalising our equivalence relation to take into account

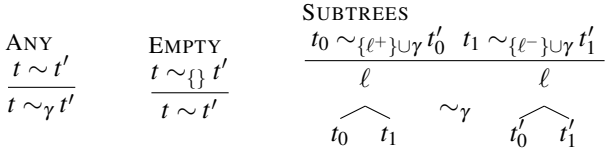


Fig. 6. Structural rules for context-aware optimisation.

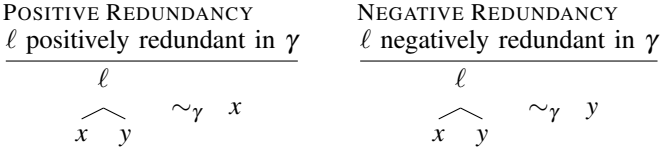


Fig. 7. Context-aware optimisation rules.

all the nodes involved in the path leading to a subtree. We call these paths *contexts* since they provide assumptions about the labels that we can consider when optimising faceted values.

Definition 8 (Contexts): A context γ is a set of *branches*, where a branch is either ℓ^+ or ℓ^- for $\ell \in \mathscr{L}$. We define the views of a context, written $\nu(\gamma)$:

$$\nu(\gamma) = \{\ell' \in \mathscr{L} \mid \forall \ell^+ \in \gamma.\ \ell \sqsubseteq \ell', \forall \ell^- \in \gamma.\ \ell \not\sqsubseteq \ell'\}$$

A context γ corresponds to a set of left (ℓ^+) or right (ℓ^-) branches taken when moving down a faceted value. The views of γ should be seen as the set of labels which can "reach" that part of the tree during projection. For example, consider the faceted value $t = \langle \text{Alice} \ ?\ \langle \text{Bob} \ ?\ 1 : 2\rangle : 3\rangle$ The labels ℓ such that $t \downarrow \ell = 2$ are precisely those ℓ such that Alice $\sqsubseteq \ell$ and Bob $\not\sqsubseteq \ell$. Equivalently, we have that $\ell \in \nu(\{\text{Alice}^+, \text{Bob}^-\})$, where the context $\{\text{Alice}^+, \text{Bob}^-\}$ encodes "taking a left" at Alice and "taking a right" at Bob.

We define equivalence up to a context, denoted $\sim_\gamma$, as follows:

Definition 9 (Equivalence up to contexts):

$$t \sim_\gamma t' \iff \forall \ell \in \nu(\gamma).\ t \downarrow \ell = t' \downarrow \ell$$

Figure 6 shows how this notion of equality relates to our earlier one ($\sim$). Rule ANY says that if faceted values are equivalent for any observer then they are equivalent under any context. Rule EMPTY shows that equivalence under an empty context corresponds to our standard equivalence notion. Furthermore, observe that equivalence under contexts subsumes our standard notion ($\sim$) since we have $t_0 \sim_{\{\}} t_1 \iff t_0 \sim t_1$ by rules ANY and EMPTY. Rule SUBTREES shows how to construct an equivalence by assuming a positive and a negative label for the left- and right-subtree, respectively. By considering contexts, we can define a natural generalisation of our notions of replacing labels.

Definition 10 (Replacement under contexts): We say that a label ℓ_2 replaces ℓ_1 under context γ, written $\ell_2 \sim_\gamma \ell_1$, if and only if $\forall \ell_o \in \nu(\gamma).\ \ell_1 \sqsubseteq \ell_o \iff \ell_2 \sqsubseteq \ell_o$
Note that this definition is more general than "replaces to the left" ($\sim^{\triangleleft}_{\ell_1}$) and "replaces to the right" ($\sim^{\triangleright}_{\ell_1}$):

- $\ell_2 \sim^{\triangleleft}_{\ell} \ell_1$ if and only if $\ell_2 \sim_{\{\ell^+\}} \ell_1$
- $\ell_2 \sim^{\triangleright}_{\ell} \ell_1$ if and only if $\ell_2 \sim_{\{\ell^-\}} \ell_1$

Naturally, if $\ell_2 \sim_\gamma \ell_1$, then $\langle \ell_2 \ ?\ x : y\rangle \sim_\gamma \langle \ell_1 \ ?\ x : y\rangle$. In what follows, we use the context information obtained when traversing the faceted tree simplify faceted values.

We start by introducing a notion of non-contradictory contexts.

Definition 11 (Coherent Context): A context γ is *coherent* if and only if $\nu(\gamma) \neq \emptyset$.

We also capture what it means for a label to be redundant in an optimisation context.

Definition 12 (Redundant labels): Label ℓ is said to be *Positively (resp. Negatively) redundant* in context γ if and only if $\forall \ell' \in \nu(\gamma).\ \ell \sqsubseteq \ell'$ (resp. $\ell \not\sqsubseteq \ell'$).
In essence, these definitions capture when a given label already has a single known relationship ($\sqsubseteq$ or $\not\sqsubseteq$) to all labels in the view induced by the context. As a result, we can introduce the rules in Figure 7. These rules gives us a way to traverse a faceted value up to a given node and decide, based on the path that took us there, if we could remove a branch. But, how do we know if a label is positively (or negatively) redundant? For that, we need to introduce the notion of *candidate label* from Schmitz et al. [12].

Definition 13 (Candidate label (from [12])): The candidate label of γ, written $\mathscr{C}(\gamma)$, is the join of all positive labels in γ, $\mathscr{C}(\gamma) = \bigsqcup\{\ \ell \mid \ell^+ \in \gamma\}$.

By applying the decision procedure of Schmitz et al. [12], we decide if a label is redundant with two simple checks.

Theorem 5 (from [12]): Given a coherent context γ, ℓ is positively redundant in γ if and only if $\ell \sqsubseteq \mathscr{C}(\gamma)$.

Theorem 6 (from [12]): Given any coherent context γ, ℓ is negatively redundant in γ if and only if $\exists \ell_1^- \in \gamma.\ \ell_1 \sqsubseteq \mathscr{C}(\gamma) \sqcup \ell$.
Schmitz et al. [12] show that coherence is a decidable property by showing that γ is coherent if and only if $\mathscr{C}(\gamma) \in \nu(\gamma)$. Finally, we get a version of Theorem 1 for context.

Theorem 7: Given ℓ that is not negatively redundant in γ, $\ell \ominus \mathscr{C}(\gamma)$ is the least label which replaces ℓ under γ.
Note that while the theorem above requires that ℓ is not negatively redundant in γ, it says nothing about positive redundancy. However, recall Theorem 5, if ℓ is positively redundant in γ, then $\ell \sqsubseteq \mathscr{C}(\gamma)$, and so $\ell \ominus \mathscr{C}(\gamma) = \bot$, which is certainly the least label which replaces ℓ under γ!

Finally, we note that the optimisation strategies presented in this section admit a natural implementation strategy, exhaustively traverse the faceted tree and "gather up" the context using the rules in Figure 6. At each label ℓ, check if it is redundant and, if so, remove it and replace it with its left or right subtree depending on if it is positively or negatively redundant. If ℓ is not redundant, replace it by $\ell \ominus \mathscr{C}(\gamma)$.

IV. COMPUTATION-ORIENTED OPTIMISATION

In this section we tie together our data-oriented optimisations from Section III with the FSME execution mode. In order to do this we introduce a *core* calculus of faceted execution designed to model the relevant parts of multi-execution necessary to show that our optimisations are semantics preserving. We also use our core language to accommodate

our novel "internalisation" of projection and the notion of focused transparency.

A. Core Calculus

We define a core language of faceted evaluation which we call λ^{Facet}. The terms of λ^{Facet} are generated by the following grammar:

$$t ::= x \mid \lambda x.\ t \mid t\ t \mid \texttt{unit} \mid \mu x.\ t \mid \langle \ell\ ?\ t : t \rangle \mid \bot$$

Where `unit` is the only value of the single-element type `unit` and $\bot$ is an element of every type. We extend the definition of projection to cover terms, with the case for facets being the same as above and all other constructs (like application and λ) are homomorphic. The full definition of projection and a number of other constructs from this section can be found in Appendix A.

Using projection, we construct an equivalence relation like the one considered in previous section, $t_0 \sim t_1$ if and only if $t_0 \downarrow \ell = t_1 \downarrow \ell$ for all $\ell \in \mathscr{L}$. Note that, while this notion of equivalence is more involved than the one for pure faceted values, the equivalences which apply to the former definitions still apply here. We formalise λ^{Facet} as a typed lambda calculus, however, the type system is standard and orthogonal to the rest of the development. We give λ^{Facet} a small-step call by name semantics. Figure 8 shows the non-standard rules, the rest can be found in Appendix A.

The rules RLEFTFACET and RRIGHTFACET allow faceted sub-computations to happen "in parallel", this is similar to the FSME execution mode of Multef [12]. The rule RFACETAPP distributes function application over facets. It is this rule which is responsible for the blowup associated with faceted values, for example $\langle \ell\ ?\ t_0 : t_1 \rangle\ \langle \ell'\ ?\ t'_0 : t'_1 \rangle \longrightarrow \langle \ell\ ?\ t_0\ \langle \ell'\ ?\ t'_0 : t'_1 \rangle : t_1\ \langle \ell'\ ?\ t'_0 : t'_1 \rangle \rangle$. Finally, the first true addition of λ^{Facet} over previous work is the REQUIV rule, which ties together the equivalences studied in Section III (through the $\sim$ equivalence) with FSME. The rule allows optimisation to happen at any point during the evaluation of a program.

Transparency: We prove that λ^{Facet} is transparent, i.e. the semantics of secure programs are not influenced by the special enforcement mechanism introduced by the faceted values. To show this, following the literature [9, 11, 12] we give a *standard semantics* $-\texttt{std}\rightarrow$ for λ^{Facet}. The role of the standard semantics is to represent "normal" evaluation of λ^{Facet} programs without enforcement in place. Using this notion we will be able to prove that λ^{Facet} programs which are secure under standard semantics do not have their semantics modified by the multi-execution enforcement mechanism. The standard semantics of λ^{Facet} are identical to the normal semantics, except that we omit the rules which deal with facets, in place of these rules we have only a single rule to remove facets, $\langle \ell\ ?\ t_0 : t_1 \rangle - \texttt{std} \rightarrow t_0$. Next we overload the $\sim$ notation, defining $t_0 \sim_\ell t_1 \iff t_0 \downarrow \ell = t_1 \downarrow \ell$. Using this definition we are going to define what it means for a program to be secure with respect to the standard semantics.

However, before we get there we need to introduce the notion of a security policy. To keep the exposition simple we only discuss the single-input programs and therefore a security policy is simply a pair (ℓ_i, ℓ) of an input label and an output label. A term t is said to be compatible with an input label ℓ_i if either t is on the form $\langle \ell_i\ ?\ t_0 : t_1 \rangle$ where t_0 is unfaceted or t itself is unfaceted. This definition of "secret input" is in line with the standard encoding of secret inputs from the multi-execution literature [9, 11, 12].

We use the notion of a secure program from [28], this definition is different from that of previous work (e.g. Multef [12]) and we refer the interested reader to Appendix A for details on why and how. In the definition below we use the standard definition of a "value", a term is a value if and only if it is either on the form $\lambda x.\ t$ or `unit`, values have the key property that they do not evaluate further.

Definition 14 (TSNI Secure Program): A term $\Gamma, x : \tau_0 \vdash t : \tau_1$ is secure w.r.t. the security policy (ℓ_i, ℓ) when, for any two inputs $\Gamma \vdash t_0, t_1 : \tau_0$ which are compatible with ℓ_i, such that $t_0 \sim_\ell t_1$, we have that if $t[t_0/x] - \texttt{std} \rightarrow^* v_0$ for some value v_0 then there exists a value v_1 such that $t[t_1/x] - \texttt{std} \rightarrow^* v_1$ and $v_0 \sim_\ell v_1$. As a consequence, if $t[t_0/x]$ does not evaluate to a value, then neither does $t[t_1/x]$.

This definition is intended to mimic previous work on faceted value semantics [9, 12, 13], where programs have input and output channels where values received on a channel are *compatible* with the label of the channel. In such a setting, the definition of secure program says that and if in two different runs the values on an *input channel* are ℓ-equivalent, where ℓ is the label of some *output channel*, then both runs generate equivalent outputs on the ℓ-labeled channel. Our definition mimics this by picking ℓ_i the label on the *input* channel and ℓ on the the *output* channel.

While the definition above is parameterised on a security policy in the form of a label pair (ℓ_i, ℓ) and a single variable $x : \tau_0$, extending it to more complex policies and multiple variables is straightforward. We direct the interested reader to the literature on faceted value semantics [9, 12, 13] for a more comprehensive treatment. Before we can address transparency we need one key lemma:

Lemma 1 (Simulation): Given an $\ell \in \mathscr{L}$ and two terms $\Gamma \vdash t, t' : \tau$ such that $t \downarrow \ell - \texttt{std} \rightarrow^* t'$ there exists an t'' such that $t \longrightarrow^* t''$ and $t' \sim_\ell t''$.

Before we address transparency we need introduce the notion of an *unfaceted* term, i.e. a term which does not use the $\langle \_\ ?\ \_ : \_ \rangle$ construct. More precisely, t is unfaceted if and only if there is some ℓ (or equivalently, for all ℓ) such that $t \downarrow \ell = t$. Note that if $t \downarrow \ell$ is equal to t, t does not contain any $\langle \_\ ?\ \_ : \_ \rangle$ subterms, as these would be removed by projection in $t \downarrow \ell$.

Theorem 8 (Transparency): For any policy (ℓ_i, ℓ), given a program $\Gamma, x : \tau_0 \vdash t : \tau_1$ which is secure with respect to (ℓ_i, ℓ) such that t is unfaceted and a term $\Gamma \vdash t_0 : \tau_0$ which is compatible with ℓ_i, we have that if $t[t_0/x] - \texttt{std} \rightarrow^* v$ for some value v, then there exists a t' such that $t[t_0/x] \longrightarrow^* t'$ and $v \sim_\ell t'$.

We prove termination sensitive noninterference (TSNI) for λ^{Facet} by following the proof strategy of earlier work [9, 12], details can be found in Appendix A.

$$\textsc{RLeftFacet}\ \frac{t_0 \longrightarrow t_0'}{\langle \ell\ ?\ t_0 : t_1\rangle \longrightarrow \langle \ell\ ?\ t_0' : t_1\rangle} \qquad \textsc{RRightFacet}\ \frac{t_1 \longrightarrow t_1'}{\langle \ell\ ?\ t_0 : t_1\rangle \longrightarrow \langle \ell\ ?\ t_0 : t_1'\rangle}$$

$$\textsc{RFacetApp}\ \langle \ell\ ?\ t_0 : t_1\rangle\ t_2 \longrightarrow \langle \ell\ ?\ t_0\ t_2 : t_1\ t_2\rangle \qquad \textsc{REquiv}\ \frac{t_0 \sim t_1}{t_0 \longrightarrow t_1}$$

Fig. 8. Small step semantics

$$\begin{aligned}
[\![\ell]\!](t_0,t_1) &= \langle \ell\ ?\ t_0 : t_1\rangle \\
[\![e \vee e']\!](t_0,t_1) &= [\![e]\!](t_0,[\![e']\!](t_0,t_1)) \\
[\![e \wedge e']\!](t_0,t_1) &= [\![e]\!]([\![e']\!](t_0,t_1),t_1) \\
[\![\neg e]\!](t_0,t_1) &= [\![e]\!](t_1,t_0)
\end{aligned}$$

$$\textsc{RProject}\ t \Downarrow e \longrightarrow [\![e]\!](t,\bot) \qquad \textsc{RProjectStd}\ t \Downarrow e - \texttt{std} \rightarrow t$$

$$(t \Downarrow e) \downarrow \ell_o = \begin{cases} t \downarrow \ell_o,\ \ell_o \models e \\ \bot,\ \text{otherwise} \end{cases}$$

Fig. 9. The semantics of $\Downarrow$

Theorem 9 (TSNI): Given any $\ell \in \mathscr{L}$ and any three well-typed terms $\Gamma \vdash t_0, t_0', t_1 : \tau$ such that $t_0 \sim_\ell t_1$ and $t_0 \longrightarrow^* t_0'$, there exists a $\Gamma \vdash t_1' : \tau$ such that $t_1 \longrightarrow^* t_1'$ and $t_0' \sim_\ell t_1'$.

Note that this theorem does not mention the notion of a policy from above, this is because the present formuation is *stronger* than a formulation which references a security policy. The theorem above works for any two ℓ-equivalent t_0 and t_1, which implies that it works for the $t[t_0/x]$ and $t[t_1/x]$ introduced by the "policy-formulation."

B. Removing unnecessary views

We now present the key computation-oriented optimisation in this paper. We show how to use knowledge of the potential observers of a faceted computation to reduce the amount of computation. We can represent such knowledge using a boolean algebra over labels:

$$e ::= \ell \mid e \vee e \mid e \wedge e \mid \neg e$$

We say that a label ℓ satisfies an expression e, written $\ell \models e$, by which we mean that the expression e predicts that ℓ is an observer of our computation. The interesting case is when $e = \ell'$, for which we have $\ell \models \ell' \iff \ell' \sqsubseteq \ell$. The rest of the definition is standard, e.g. $\ell \models e_0 \wedge e_1$ if and only if $\ell \models e_0$ and $\ell \models e_1$, and can be found in Appendix A.

The idea behind label expressions is that if $\ell \models e$, then the term $t \Downarrow e$ can be used in a computation which will be written to an output channel with label ℓ. In Section V we will see how this can be used to keep the number of sub-computations introduced by faceted value semantics tractable in a setting with multiple input and output channels. For now, however, we will focus on the single-input single-output case of λ^{Facet} in order to keep the formalism concise.

We give a semantics to label expressions in terms of faceted values, written $[\![e]\!](t_0,t_1)$, which can be found in Figure 9. The definition has the property that $[\![e]\!](t_0,t_1) \downarrow \ell = t_0 \downarrow \ell$ when $\ell \models e$ and $t_1 \downarrow \ell$ otherwise. Next we extend our faceted language with a primitive $\Downarrow$ which models injecting knowledge about the future observers of the computation into our program.

$$\frac{\ell \models e \quad \ell \succeq t}{\ell \succeq t \Downarrow e} \qquad \frac{\ell' \sqsubseteq \ell \quad \ell \succeq t_0}{\ell \succeq \langle \ell'\ ?\ t_0 : t_1\rangle}$$

$$\ell \models \ell' \iff \ell' \sqsubseteq \ell$$

Fig. 10. Non-structural cases of $\ell \succeq t$ and $\ell \models e$

$$t ::= \ldots \mid t \Downarrow e$$

For the semantics, we introduce the RProject and RProjectStd rules in Figure 9.

There are two important things to note about the semantics of $\Downarrow$. The first is that while we call the rule RProject and use a syntax similar to projection, what we are doing is not strictly projection, we do not remove parts of the faceted value, only encode additional assumptions. This choice is motivated by the fact that internalising projection as something similar to $t \Downarrow \ell = \langle \ell\ ?\ t \downarrow \ell : \bot\rangle$ breaks noninterference, at least when noninterference is formulated in the traditional way. To see why, consider the H-equivalent terms $(\lambda x.x \Downarrow \texttt{L})\ \langle \texttt{H}\ ?\ 1 : 0\rangle$ and $(\lambda x.x \Downarrow \texttt{L})\ \langle \texttt{H}\ ?\ 1 : 1\rangle$, which would reduce to 0 and 1 respectively, which are not H-equivalent. The second thing to note is that while we have chosen to present the meaning of $\Downarrow$ in terms of dynamic semantics, it could also be formulated as a static "compilation step." Our choice is motivated the fact that the dynamic $\Downarrow$ more closely mimics our implementation in Section V.

The $\Downarrow$ optimisation does not come for free. Recall that the condition in the transparency theorem from above is that the source program is both secure and unfaceted, that is $t \downarrow \ell = t$. This condition is not fulfilled for programs containing $\Downarrow$. Consequently, while including $\Downarrow$ in the calculus does not break noninterference or transparency, as soon as we *use* $\Downarrow$, we no longer have transparency guarantees.

To address this issue, we introduce a notion of *focused transparency* which subsumes the traditional notion from above. The intuition behind focused transparency is that if the $\Downarrow$ annotations are correct, then we do not modify the semantics of secure programs when running under faceted as opposed to standard semantics. To formalise the idea of correct annotations we introduce the notion of "clearance." Intuitively, a label ℓ clears t, written $\ell \succeq t$, if ℓ satisfies all the label expressions e in t, and it can see all the facets in t. In other words, $\ell \succeq t$ if ℓ is one of the intended observers of t, i.e. the view of t associated to ℓ is not culled by any $\Downarrow$ optimisation. Figure 10 shows the interesting (non-structural) cases of $\ell \succeq t$,

the rest can be found in Appendix A.

With clearance in place, we can move on to proving focused transparency. The proof of focused transparency is similar to the proof of normal transparency, but here we generalise the definition to include faceted terms t where $\ell \succeq t$ and show that the theorem still holds. In this presentation we omit some tedious lemmas to do with the interaction between projection and clearing. We note only only that the proof of the next theorem relies on a version of the Simulation lemma with precondition $\ell \succeq t$, the interested reader will find details in Appendix A.

Theorem 10 (Focused Transparency): For any policy (ℓ_i, ℓ), given a program $\Gamma, x : \tau_0 \vdash t : \tau_1$ which is secure with respect to (ℓ_i, ℓ) such that $\ell \succeq t$ (i.e. the annotations in t are correct) and a term $\Gamma \vdash t_0 : \tau_0$ which is compatible with ℓ_i, we have that if $t[t_0/x] - \texttt{std} \rightarrow^* v$ for some value v, then there exists a t' such that $t[t_0/x] \longrightarrow^* t'$ and $v \sim_\ell t'$.

As $\ell \succeq t \downarrow \ell'$ holds for all ℓ and ℓ', focused transparency implies traditional transparency as presented in Theorem 8. In other words, our notion of focused transparency is can safely replace the traditional notion from above.

V. Case Studies

In this section, we present two case studies to show how the optimisation techniques in this paper can be applied to the multi-execution framework Multef [12].

A. Data-Oriented Optimisation

Our data-oriented optimisations are suitable when dealing with storage of faceted values. For example, Yang et al. [16] use faceted values in a database backed application to encode multiple views of the same data. Intuitively, each "leaf" of the faceted value gets stored as a record within a table—the more leaves the faceted values has, the more space it takes in the database. One of the case studies of Yang et al. presents a small calendar application where faceted values provide different views of calendar events. In their model, group events can have sophisticated views associated with them which allows encoding multiple groups of event attendees with different privileges w.r.t. what they can observe. For example, if Bob and Charlie are organising an event for Alice, the calendar event may be encoded as the faceted value in Figure 11. Where v_0 may contain secret information like what gifts Bob and Charlie are planning to give Alice, while v_1 encodes what Alice gets to see, i.e., that she has an event next Wednesday. Plainly following Yang et al.'s approach for storing events in the database will likely, considering that faceted values might blow up in size when combined with other faceted values, result in storing a prohibitively large number of records in the database, especially after computing *aggregates* of data.

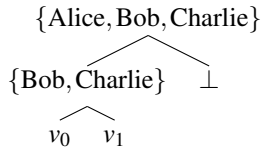


Fig. 11. Example for optimising

Employing the data oriented-optimisations presented in this paper, however, allows us to shrink these combined faceted values to a manageable size. Our experiment considers a simple implementation of our shrinking optimisations. We apply the Join rule exhaustively, then we apply the context optimisations from Figure 7 and Figure 6 alongside Choice Irrelevance and residuation with the candiate label, finally we rotate the tree back (to balance it) using Qualified Rotation and exhaustively apply Choice Irrelevance again. This approach is not optimal in terms of speed, it runs in (at least) quadratic time. However, for the purpose of this case study we focus on the storage overhead of faceted values and therefore we are not concerned with optimising or precisely measuring time overheads.

To show the effect of applying our shrinking techniques, we randomly and uniformly generated two sets of 10^5 lists of four faceted values, one with integer leaves and the other with leaves of pairs of integers representing the start and end time of a meeting. The events were between 1 and 4 hours long and all scheduled during business hours on the same day. Each faceted value has two or three different leaves with labels drawn from the powerset lattice over 15 different principals – so, the lattice contains 2^{15} possible points. In a real application, these faceted values could be derived from four calendar events.

We then compute the aggregation of the faceted values by taking their sum (integer leaves) and computing if any meetings overlap (meeting leaves) respectively. Finally, we compare three different distributions of sizes.

We first consider the case for computing sums. The first distribution (`UNOPT`) is the size of the faceted values in the number of leaves without optimising. The second (`OPT1`) is the result of optimising each value before computing the sum. Finally, the third distribution (`OPT2`) is the result of optimising both before and after computing the sum. Figure 12 shows the cumulative distributions. The x axis shows the number of leaves in the resulting trees and the y axis shows the cumulative number of occurrences. The value of y at x shows how many faceted values have fewer or equal to x number of leaves.

The results in Figure 12 show that naively computing aggregates quickly leads to blow up in faceted value sizes. The largest trees in `UNOPT` have 81(!) leaves with an average of 39.1 leaves per tree, the same goes for `OPT1` but with an average of 21 leaves, while `OPT2` has a maximum of 48 and average 10.7 leaves! The `OPT1` distribution shows that optimising the faceted values reduces the amount of blowup, the difference between `OPT1` and `OPT2` makes it clear that optimisation needs to be performed continuously as aggregation and computation themselves introduce redundant views of faceted values.

Turning our attention to the distribution of sizes for the meeting times in Figure 13 we see more extreme results than in the other experiment. The largest trees in `OPT2` have only 13 leaves and with an average of 1.1! This is because the Choice Irrelevance rule kicks in more often, as there are only two possible values for the leaves after computing the

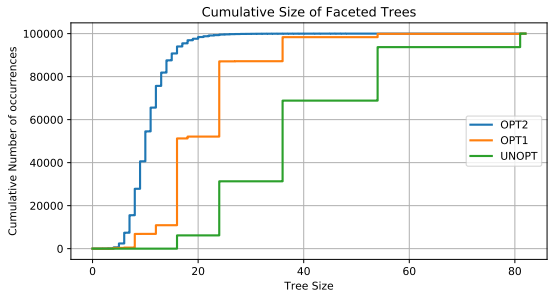


Fig. 12. Data-optimisations on faceted values (computing sums)

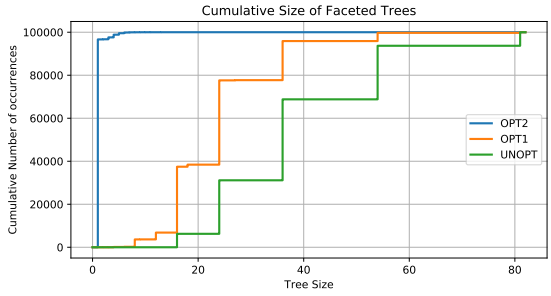


Fig. 13. Data-optimisations on faceted values (computing overlapping times)

overlap (True and False). Interestingly, the `OPT1` and `UNOPT` results look very similar to the case for sums. We attribute this similarity to the fact that optimising only before computing the sums means that the optimisation opportunities are mostly due to the label structure, not the values at the leaves.

B. Computation-Oriented Optimisation

For this case study, we present a small chat server called `FChat`, written in the multi-execution framework Multef [12]. The server allows clients to connect to channels where they can send and receive messages. Figure 14 shows the relevant Haskell [29] code of the server's main loop.

The security policy of `FChat` is expressed with DC-labels. Each chat channel is given a DC-label $(c_0^r \vee c_1^r \vee \ldots \vee c_n^r, c_0^w \vee c_1^w \vee \ldots \vee c_k^w)$, where c_i^r are the clients allowed to read the channel and c_i^w the clients allowed to write to it. The actual implementation also needs to consider declassification and endorsement of messages to be able conform to this security policy. However, this is mostly orthogonal to the rest of this discussion and these details are omited here in the interest of clarity.

```
mainLoop :: Policy -> FIO ()
mainLoop pol = do
  fevent <- readFIChan evCh
  fstate <- readFIORef stRef
  control $ do
    st <- fstate
    ev <- fevent
    return $ case ev of
      Connect client chan
        | canConnect pol client chan -> do
          write stRef ((client, chan) : st)

      Write client chan msg
        | canWrite pol client chan -> do
          sequence_ [ clientWrite client msg
                    | (client, ch) <- st
                    , chan == ch ]

  ...
```

Fig. 14. The main loop of the unoptimised chat server

Primitive `readFIChan :: IChan a -> FIO (Faceted a)` reads a faceted event from a channel—variable `fevent` in the code. In the interest of simplicity, the trusted computing based (TCB) around `FChat` is responsible for parsing client messages and generating faceted events accordingly. For instance, if Alice connects to the channel `chan`, the TCB places a faceted event $\langle$`Alice ? Connect Alice chan :` $\bot\rangle$ in the event channel.

The code in Figure 14 keeps the state of all connections in a reference `stRef`. The value stored in that reference (`fstate`) is also faceted, as the state of alive connections depends on who the observer is—recall that only Alice will see that she has an open connection. The `control` primitive executes the code that follows for every leaf in the "cartesian product" of `fevent` and `fstate`—this is the primitive that introduces multi-executions in Multef. Inside the `control` block, variables `st` and `ev` are leaves in `fstate` and `fevent`, respectively.

When it comes to connecting users to channels, the main loop simply checks that principals appear in the DC-label of the channel (see check `canConnect pol client chan`). In that case, that faceted state of the server gets updated to reflect the new connection (`write stRef ((client, chan) : st)`).

Observe that after n clients have connected, without even sending any messages, the faceted state of the server `stRef` becomes a reference containing a faceted value with 2^n leaves! The exponential size of `fstate` means that when a new connection arrives or a message is sent to the server, the code inside `control` is executed $O(2^n)$ times! This exponential complexity is in contrast to the $O(n)$ time complexity which would be required if the server was running under a standard, but insecure, semantics instead of a faceted one. The degradation is reflected in the exponential runtime of `UNOPT` in our experiment, shown in Figure 15.

The key observation that enables performance optimizations in this case study is that, while the `control` block runs an exponential number of times, most of the instructions are no-ops. To see why, recall that events arising from reading channels are of the form $\langle p \,?\, \text{event} : \bot\rangle$ for some principal p and event, e.g., writing to a channel. In this light, and since Multef guarantees security and transparency, the main loop is guaranteed to send messages to clients in the same way as under an standard semantics. Hence, from all the $O(2^n)$ times Multef executes the `control` block, only one will trigger the side-effect of writing to a channel. More specifically, when using the side-effectful primitive `clientWrite`, Multef checks that the current view is the one in which the side-effect can be triggered (i.e. the one compatible with the DC-label of

the channel)—otherwise it behaves as a no-op. The security policy of our application considers each client c's channel as labeled $(c, \top)$, i.e., we are only able to write to it if we have not branched on any information more secret than c. This means that we only need to consider the parts of the faceted state `fstate` which contain only c's private branches!

With this in mind, we can employ the $\Downarrow$ optimisation to reduce the exponential blowup to consider only these leaves of the faceted trees! We implement $\Downarrow$ as a function `atMostOne :: [Client] -> Faceted a -> Faceted a`. The expression `atMostOne ["Alice", "Bob"] f` should be read as $f \Downarrow (\text{Alice} \wedge \neg\text{Bob} \vee \neg\text{Alice} \wedge \text{Bob})$. We use `atMostOne` to optimise away the unnecessary leaves of the state:

```
mainLoop :: Policy -> FIO ()
mainLoop pol = do
  st0 <- readFIORef stRef
  let state = atMostOne (principals pol) st0
  writeFIORef stRef state
  ...
```

The rest of the code (denoted by `...`) is the same as above. Over time, the content of the reference `stRef` increases in size with extra faceted constructors due to the command `writeFIORef stRef state`. Consequently, if we do nothing, `st0` will grow exponentially large. For this reason, the function `atMostOne` also employs the $\sim$ optimisation techniques from Section III to remove redundant labels in the tree denoted by `st0`. With this optimisation in place, we see an important difference in performance.

Figure 15 shows the time taken for N clients to connect to the server and send 10 messages each to a single shared channel. The unoptimised version of the code, shown in the figure as `UNOPT`, is exponential in N, whereas the optimised implementation runs in polynomial time. In our particular example the memory behaviour of the optimised an unoptimised versions of the program are practically identical. We attribute this phenomenon to the small number of connected clients, limited to 10 due to the exponential blowup in the unoptimised implementation, which means that the overhead in memory is relatively small and difficult to measure.

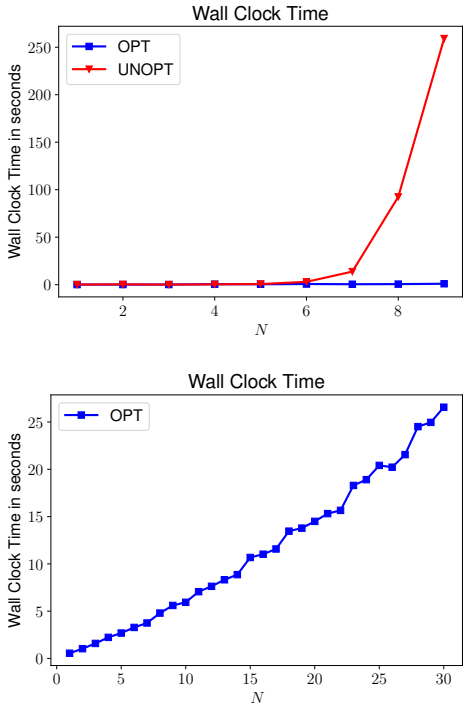


Fig. 15. The Performance of the Optimised and Unoptimised implementations of the chat server.

VI. Related Work

Much prior work has explored IFC techniques based on multi-execution [9, 11, 12, 13], running multiple copies of a program (or parts of it) simultaneously at different security levels, while carefully adapting their semantics to avoid information leakage. Capizzi et al. [30] propose running two copies of the same program, one for public and other for private data. Cristiá and Mata independently formalize a similar system at the operating system level [31]. Devriese and Piessens [32] coin the term Secure Multi-Execution (SME) and are the first to formalise the soundness and precision guarantees of the approach. This original formulation of SME is *black-box*, i.e, language independent, which makes it possible to deploy it for complex languages like JavaScript. Jaskelioff and Russo [33] present an implementation of SME in Haskell. Barthe et al. [34] propose a program that inlines SME into JavaScript-like programs—so that it is not necessary to modify the runtime system to obtain multi-executions. The web has been the chosen domain to test many SME ideas [35] and their implementations, e.g., FlowFox [36]. SME has also been applied to the map-reduce programming model [37].

Secure programs interpreted under SME produce the same outputs as if they were run under a standard semantics *modulo the relative ordering of observable events from different security levels*. The work in [38] explores how different scheduling policies affect the security guarantees provided by SME, i.e., TINI or TSNI. In [39, 40], the authors combine scheduling techniques with monitoring approaches to guarantee that interleaving of events gets preserved for secure programs. The authors of [40, 41, 42] provide means for declassification.

A limitation of SME is that it requires multiple executions, potentially one for each element in the security lattice, which is particularly problematic for large powerset lattices or unbounded decentralized lattices. To address this concern, Austin and Flanagan introduce a Multiple Facets (MF) semantics [43] as an optimization for SME. Schmitz et al. [19] show an implementation of MF in Haskell. Schoepe et al. [44] investigate how to apply MF semantics to encode taint analysis. Bielova and Rezk [15] later show that SME and MF provide different security guarantees, namely TINI for MF vs. TSNI for SME. They propose an all-or-nothing combination of MF and SME that runs programs under a MF semantics but switches to SME when commands inside a branch do not terminate within a timeout. In the same all-or-nothing spirit, Ngo et al. [18] combine MF and SME for a simple while-language, where timeouts determine when to switch to SME.

That work also presents some local "peephole" optimizations for faceted values that are applied when values are constructed, rather than at arbitrary times during execution as in our work. Schmitz et al. [12] present a synthesis of MF and SME called Faceted Secure Multi-Execution (FSME), and an underlying multi-execution framework in Haskell called Multef that can be parameterized to provide either MF, SME, or FSME.

Yang et al. [16] present an approach for persisting faceted values into a database, and include an "early pruning" optimization that shrinks faceted values according to the observer, in a manner that is similar to our projection operator. A key distinction is that our projection operation is a construct in the language, thus enabling the programmer to control where this simplification is applied.

Many other IFC security libraries exists for Haskell. They can enforce non-interference statically [45, 46, 47, 48], dynamically [7], or as a combination of both [49, 50]. Many of these libraries utilize the concept of monads to control the side-effects that programs are allowed to perform.

VII. Conclusions

We have present both data- and computation-oriented optimisations for multi-execution based techniques like MF, "on-demand" SME (as implemented by [12, 13]), and FSME. The data-oriented optimisations are based on tree transformations which preserve the *views* of faceted values. Meanwhile, the computation-oriented optimisations allow programmers to reduce unnecessary computations by making assumptions about who observes the produced results. We provided case studies that support our claims. As future work, we plan to explore different rewriting strategies and evaluate the trade-offs between the time they take to execute and the introduced space savings—an aspect that our case studies do not evaluate.

We believe that this work fortifies the foundations of more practical multi-execution based techniques for realistic scenarios, where we not only optimisations are needed but also our novel notion of *focused transparency*.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful and insightful comments. This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011), and the Swedish research agency Vetenskapsrådet.

References

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, pp. 5–19, Sep. 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2002.806121

[2] J. Goguen and J. Meseguer, "Security policies and security models," in *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982.

[3] D. Volpano, G. Smith, and C. Irvine, "A Sound Type System for Secure Flow Analysis," *J. Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[4] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java Information Flow," 2001, http://www.cs.cornell.edu/jif.

[5] M. Vassena, A. Russo, P. Buiras, and L. Waye, "Mac a verified static information-flow control library," *Journal of Logical and Algebraic Methods in Programming*, vol. 95, pp. 148 – 180, 2018.

[6] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1663–1671.

[7] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*, 2011.

[8] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS '10. ACM, 2010.

[9] ——, "Multiple facets for dynamic information flow," in *ACM Sigplan Notices*, vol. 47, no. 1. ACM, 2012, pp. 165–178.

[10] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Cant live with em, cant live without em," in *International Conference on Information Systems Security*. Springer, 2008, pp. 56–70.

[11] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 109–124.

[12] T. Schmitz, M. Algehed, C. Flanagan, and A. Russo, "Faceted secure multi execution," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 1617–1634. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243806

[13] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, "A better facet of dynamic information flow control," in *WWW'18 Companion: The 2018 Web Conference Companion*, 2018, pp. 1–9.

[14] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *Nordic conference on secure IT systems*. Springer, 2011, pp. 223–239.

[15] N. Bielova and T. Rezk, "Spot the difference: Secure multi-execution and multiple facets," in *European Symposium on Research in Computer Security*, 2016, pp. 501–519.

[16] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 631–647.

[17] M. Ward and R. P. Dilworth, "Residuated lattices," *Trans-*

actions of the American Mathematical Society, vol. 45, no. 3, pp. 335–354, 1939.

[18] M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz, "A better facet of dynamic information flow control," in *The Web Conference. Research track: Security and privacy of the Web. (WWW'18)*, 2018.

[19] T. Schmitz, D. Rhodes, T. H. Austin, K. Knowles, and C. Flanagan, "Faceted dynamic information flow via control and data monads," in *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, 2016.

[20] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *Proc. of the Nordic Conference on Information Security Technology for Applications (NORDSEC '11)*. Springer-Verlag, 2011.

[21] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[22] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *Nordic conference on secure IT systems*. Springer, 2011.

[23] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications." in *OSDI*, 2012, pp. 47–60.

[24] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining JavaScript with COWL," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Oct. 2014.

[25] J. Parker, N. Vazou, and M. Hicks, "Lweb: information flow security for multi-tier web applications," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 75, 2019.

[26] B. Montagu, B. Pierce, and R. Pollack, "A theory of information-flow labels," in *Computer Security Foundations Symposium (CSF), 2013 IEEE 26th*, June 2013.

[27] M. Erné, J. Koslowski, A. Melton, and G. E. Strecker, "A primer on galois connections," *Annals of the New York Academy of Sciences*, vol. 704, no. 1, pp. 103–125, 1993.

[28] D. Hedin and A. Sabelfeld, "A Perspective on Information-Flow Control," in *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

[29] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson, "Report on the programming language haskell: A non-strict, purely functional language version 1.2," *SIGPLAN Not.*, vol. 27, no. 5, pp. 1–164, May 1992. [Online]. Available: http://doi.acm.org/10.1145/130697.130699

[30] R. Capizzi, A. Longo, V. N. Venkatakrishnan, and A. P. Sistla, "Preventing Information Leaks through Shadow Executions," in *Proc. of the Annual Computer Security Applications Conference*, ser. ACSAC '08. IEEE Computer Society, 2008.

[31] M. Cristiá and P. Mata, "Runtime Enforcement of Noninterference by Duplicating Processes and their Memories," in *Workshop de Seguridad Informática WSEGI 2009, Argentina*, ser. 38 JAIIO, 2009.

[32] D. Devriese and F. Piessens, "Noninterference through Secure Multi-execution," in *Proc. of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. IEEE Computer Society, 2010.

[33] M. Jaskelioff and A. Russo, "Secure multi-execution in Haskell," in *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, ser. LNCS. Springer-Verlag, Jun. 2011.

[34] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas, "Secure multi-execution through static program transformation," in *Formal Techniques for Distributed Systems (FMOODS/FORTE 2012)*, June 2012.

[35] N. Bielova, D. Devriese, F. Massacci, and F. Piessens, "Reactive non-interference for a browser model," in *Proceedings of the 5th International Conference on Network and System Security (NSS 2011),*, Sep. 2011.

[36] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: a web browser with flexible and precise information flow control," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012.

[37] M. Ngo, F. Massacci, and O. Gadyatskaya, "MAP-REDUCE runtime enforcement of information flow policies," *CoRR*, 2013. [Online]. Available: http://arxiv.org/abs/1305.2136

[38] V. Kashyap, B. Wiedermann, and B. Hardekopf, "Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach," in *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[39] D. Zanarini, M. Jaskelioff, and A. Russo, "Precise enforcement of confidentiality for reactive systems." in *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE, 2013, pp. 18–32.

[40] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," in *2013 IEEE 26th Computer Security Foundations Symposium*, June 2013, pp. 33–48.

[41] M. Vanhoef, W. De Groef, D. Devriese, F. Piessens, and T. Rezk, "Stateful declassification policies for event-driven programs," in *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE, Jul. 2014.

[42] I. Boloşteanu and D. Garg, "Asymmetric secure multi-execution with declassification," in *Principles of Security and Trust*, F. Piessens and L. Viganò, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 24–45.

[43] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Princi-*

$$
\begin{aligned}
\ell \models \ell' &\iff \ell' \sqsubseteq \ell \\
\ell \models e \vee e' &\iff \ell \models e \text{ or } \ell \models e' \\
\ell \models e \wedge e' &\iff \ell \models e \text{ and } \ell \models e' \\
\ell \models \neg e &\iff \ell \not\models e
\end{aligned}
$$

Fig. 16. The full definition of $\ell \models e$

ples of programming languages, ser. POPL '12. ACM, 2012.

[44] D. Schoepe, M. Balliu, F. Piessens, and A. Sabelfeld, "Let's face it: Faceted values for taint tracking," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, 2016.

[45] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical Computer Science*, vol. 411, no. 19, pp. 1974–1994, 2010.

[46] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in Haskell," in *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM, Sep. 2008.

[47] M. Vassena, A. Russo, P. Buiras, and L. Waye, "Mac a verified static information-flow control library," *Journal of Logical and Algebraic Methods in Programming*, 2017.

[48] M. Algehed and A. Russo, "Encoding DCC in Haskell," in *Proc. of the 2017 Workshop on Programming Languages and Analysis for Security*, ser. PLAS '17. ACM, 2017.

[49] D. Devriese and F. Piessens, "Information flow enforcement in monadic libraries," in *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*. ACM, 2011.

[50] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, 2015.

APPENDIX A
COMPUTATION ORIENTED OPTIMISATIONS

In this appendix we provide full definitions of semantics and operations as well as proof sketches for the main theorems of Section IV. Full proofs can be found in the Agda mechanisation. Figure 17 shows the full semantics of λ^{Facet}. Figure 16 contains the full definition of $\ell \models e$.

Our definition of a secure program differs from that of both Schmitz et al. [12] and Devriese and Piessens [11] because their notions are too restrictive to accurately capture what it means to be a secure program. We will demonstrate this by considering the program $\lambda x.\texttt{if } x \texttt{ then } 1 \texttt{ else } 1$ with the security policy $(\texttt{H},\texttt{L})$, secret input and public output. In [12], t is secure if when t_0 and t_1 are low equivalent and $t\ t_0 - \texttt{std} \rightarrow^* t'$, there exists a term t'' such that $t\ t_1 - \texttt{std} \rightarrow^* t''$ and t' is low equivalent to t''. The definition requires that all of the reduction steps for $t\ t_0$ can be mimicked by $t\ t_1$. This is best illustrated by comparing the evaluation of $t = \lambda x.\texttt{if } x \texttt{ then } 1 \texttt{ else } 1$ with inputs $t_0 = \langle \texttt{H} \ ?\ 0 : 0 \rangle$ and $t_1 = \langle \texttt{H} \ ?\ 1 : 0 \rangle$. Both $t\ t_0$ and $t\ t_1$ terminate with the value 1 and t is clearly a secure program. The problem is that $t\ t_0 - \texttt{std} \rightarrow^* \texttt{if } 1 \texttt{ then } 1 \texttt{ else } 1$ before terminating, and in order for t to be secure in the sense of [12], they also require that $t\ t_1 - \texttt{std} \rightarrow^* \texttt{if } 1 \texttt{ then } 1 \texttt{ else } 1$. Because the latter

$$
\frac{t_0 \longrightarrow t_0'}{t_0\ t_1 \longrightarrow t_0'\ t_1}\ \textsc{RAppCong} \qquad (\lambda x.\ t_0)\ t_1 \longrightarrow t_0[t_1/x]\ \textsc{RBeta}
$$

$$
\mu x.\ t \longrightarrow t[\mu x.\ t/x]\ \textsc{RFix} \qquad \frac{t_0 \longrightarrow t_0'}{\langle \ell \ ?\ t_0 : t_1 \rangle \longrightarrow \langle \ell \ ?\ t_0' : t_1 \rangle}\ \textsc{RLeftFacet}
$$

$$
\frac{t_1 \longrightarrow t_1'}{\langle \ell \ ?\ t_0 : t_1 \rangle \longrightarrow \langle \ell \ ?\ t_0 : t_1' \rangle}\ \textsc{RRightFacet}
$$

$$
\langle \ell \ ?\ t_0 : t_1 \rangle\ t_2 \longrightarrow \langle \ell \ ?\ t_0\ t_2 : t_1\ t_2 \rangle\ \textsc{RFacetApp} \qquad \frac{t_0 \sim t_1}{t_0 \longrightarrow t_1}\ \textsc{REquiv}
$$

$$
t \Downarrow e \longrightarrow [\![e]\!](t, \bot)\ \textsc{RProject} \qquad \bot\ t \longrightarrow \bot\ \textsc{Bottom}
$$

Fig. 17. The full operational semantics of λ^{Facet}

$$
\begin{aligned}
x \downarrow \ell_o &= x \\
\lambda x.\ t \downarrow \ell_o &= \lambda x.\ (t \downarrow \ell_o) \\
t_0\ t_1 \downarrow \ell_o &= (t_0 \downarrow \ell_o)\ (t_1 \downarrow \ell_o) \\
\texttt{unit} \downarrow \ell_o &= \texttt{unit} \\
\mu x.\ t \downarrow \ell_o &= \mu x.(t \downarrow \ell_o) \\
\langle \ell \ ?\ t_0 : t_1 \rangle \downarrow \ell_o &= \begin{cases} t_0 \downarrow \ell_o, & \ell \sqsubseteq \ell_o \\ t_1 \downarrow \ell_o, & \text{otherwise} \end{cases} \\
(t \Downarrow e) \downarrow \ell_o &= \begin{cases} t \downarrow \ell_o, & \ell_o \models e \\ \bot, & \text{otherwise} \end{cases} \\
\bot \downarrow \ell_o &= \bot
\end{aligned}
$$

Fig. 18. The full definition of projection extended to λ^{Facet}

$$
\ell \succeq x \qquad \frac{\ell \succeq t}{\ell \succeq \lambda x.\ t} \qquad \frac{\ell \succeq t_0 \quad \ell \succeq t_1}{\ell \succeq t_0\ t_1} \qquad \ell \succeq \texttt{unit}
$$

$$
\frac{\ell \succeq t}{\ell \succeq \mu x.\ t} \qquad \ell \succeq \bot \qquad \frac{\ell \models e \quad \ell \succeq t}{\ell \succeq t \Downarrow e} \qquad \frac{\ell' \sqsubseteq \ell \quad \ell \succeq t_0}{\ell \succeq \langle \ell' \ ?\ t_0 : t_1 \rangle}
$$

Fig. 19. Full definition of $\ell \succeq t$

is not the case, t is not secure by the definition in [12]. In order to avoid this issue, we instead use the notion of a secure program from [28]:

Definition 15 (TSNI Secure Program): A term $\Gamma, x : \tau_0 \vdash t : \tau_1$ is secure w.r.t. the security policy (ℓ_i, ℓ) when, for any two inputs $\Gamma \vdash t_0, t_1 : \tau_0$ which are compatible with ℓ_i, such that $t_0 \sim_\ell t_1$, we have that if $t[t_0/x] - \texttt{std} \rightarrow^* v_0$ for some value v_0 then there exists a value v_1 such that $t[t_1/x] - \texttt{std} \rightarrow^* v_1$ and $v_0 \sim_\ell v_1$. As a consequence, if $t[t_0/x]$ does not evaluate to a value, then neither does $t[t_1/x]$. We also state the definition diagrammatically in Figure 20.

Full lines in the diagram in Figure 20, and the diagrams to come, denote preconditions and dashed lines denote the "result", what is in the existential quantification. For example, in the definition above $t[t_0/x] - \texttt{std} \rightarrow^* v_0$ is given, while v_1 and $t[t_1/x] - \texttt{std} \rightarrow^* v_1$ are obtained from the existential.

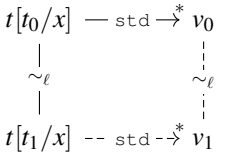
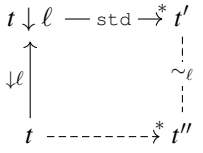
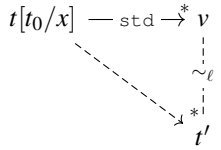


Fig. 20. TSNI Secure Program

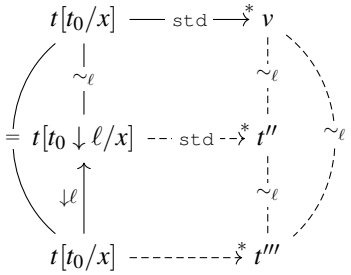
Lemma 2 (Simulation): Given an $\ell \in \mathscr{L}$ and two terms $\Gamma \vdash t, t' : \tau$ such that $t \downarrow \ell - \texttt{std} \rightarrow^* t'$ there exists an t'' such that $t \longrightarrow^* t''$ and $t' \sim_\ell t''$.



Theorem 11 (Transparency): For any policy (ℓ_i, ℓ), given a program $\Gamma, x : \tau_0 \vdash t : \tau_1$ which is secure with respect to (ℓ_i, ℓ) such that $t \downarrow \ell = t$, that is to say that t is unfaceted, and a term $\Gamma \vdash t_0 : \tau_0$ which is compatible with ℓ_i. We have that if $t[t_0/x] - \texttt{std} \rightarrow^* v$ for some value v, then there exists a t' such that $t[t_0/x] \longrightarrow^* t'$ and $v \sim_\ell t'$. Stated as a diagram:

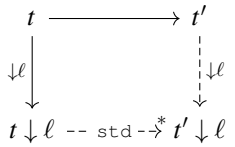


Proof: We give a diagrammatic proof sketch:

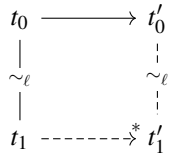


The top square, relating the standard evaluation of $t[t_0/x]$ with that of $t[t_0 \downarrow \ell/x]$ is obtained from the definition of a secure program. The bottom square is obtained from the Simulation lemma. Note that this lemma requires $t[t_0/x] \downarrow \ell - \texttt{std} \rightarrow^* t''$, but we have $t[t_0 \downarrow \ell/x] - \texttt{std} \rightarrow^* t''$. This is where our condition that $t \downarrow \ell = t$ comes in, as for any t_0, t_1, ℓ we have $t_0[t_1/x] \downarrow \ell = t_0 \downarrow \ell[t_1 \downarrow \ell/x]$. The bottom reduction, $t[t_0/x] \longrightarrow^* t'''$ is the required diagonal in the statement of the theorem. ■

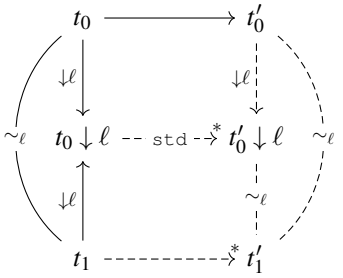
Lemma 3 (Projection): Given any $\ell \in \mathscr{L}$ and any two well typed terms $\Gamma \vdash t, t' : \tau$ such that $t \longrightarrow t'$ we have that $t \downarrow \ell - \texttt{std} \rightarrow^* t' \downarrow \ell$ in zero or one steps. When rendered diagrammatically we have:



Lemma 4 (Single Step TSNI): Given any $\ell \in \mathscr{L}$ and any three well-typed terms $\Gamma \vdash t_0, t_0', t_1 : \tau$ such that $t_0 \sim_\ell t_1$ and $t_0 \longrightarrow t_0'$, there exists a $\Gamma \vdash t_1' : \tau$ such that $t_1 \longrightarrow^* t_1'$ and $t_0' \sim_\ell t_1'$. Rendered in diagram form, we have:



Proof: We chase diagrams:



The top square is obtained from the Projection lemma and the bottom from Simulation. ■

Theorem 12 (TSNI): Given any $\ell \in \mathscr{L}$ and any three well-typed terms $\Gamma \vdash t_0, t_0', t_1 : \tau$ such that $t_0 \sim_\ell t_1$ and $t_0 \longrightarrow^* t_0'$, there exists a $\Gamma \vdash t_1' : \tau$ such that $t_1 \longrightarrow^* t_1'$ and $t_0' \sim_\ell t_1'$.

Proof: By induction on the derivation of $t_0 \longrightarrow^* t_0'$, making use of Single Step TSNI in the step case. ■

Theorem 13 (Focused Transparency): For any policy (ℓ_i, ℓ), given a program $\Gamma, x : \tau_0 \vdash t : \tau_1$ which is secure with respect to (ℓ_i, ℓ) such that $\ell \succeq t$ (i.e. the annotations in t are correct) and a term $\Gamma \vdash t_0 : \tau_0$ which is compatible with ℓ_i, we have that if $t[t_0/x] - \texttt{std} \rightarrow^* v$ for some value v, then there exists a t' such that $t[t_0/x] \longrightarrow^* t'$ and $v \sim_\ell t'$.

The full proof, which can be found in the Agda mechanisation, works by the same diagram chasing style argument as the proof of transparency above.