LIO⋆: Low Level Information Flow Control with F⋆

JEAN-JOSEPH MARTY, INRIA, IRISA, France
LUCAS FRANCESCHINO, INRIA, IRISA, France
JEAN-PIERRE TALPIN, INRIA, IRISA, France
NIKI VAZOU, IMDEA Software Institute, Spain

We present Labeled Input-Outputs in F⋆ (LIO⋆), a verified library to enforce information flow control (IFC) policies for embedded programs designed in F⋆. Our library encapsulates IFC policies into effects, allows IFC runtime checks in source F⋆ programs to be lifted by static proof obligations, supports client-wise non-interference proofs using metaprogramming, and automatically extracts to 0-overhead efficient, low-level, and provably correct C code. We benchmark our library on three client programs and observe up to 54% speedup when IFC runtime checks are proved statically. Our library is designed to aid development of embedded devices where both enforcement of security policies and efficient low-level code is critical.

Additional Key Words and Phrases: verified programming, refinement types, embedded devices, information flow control

1 INTRODUCTION

Embedded devices are part of connected environments that bring wisdom to systems (*e.g.,* smart cars, house appliances, grids) by sensing, processing and broadcasting information from multiple sources. Such devices are programmed in low-level system languages, like C, to form components, *e.g.,* bus or GPS, that need to be efficient and resource-constrained. As this components become growingly part of interconnected systems, it is critical to enforce end-to-end privacy and security policies, i.e. component separation.

Information Flow Control (IFC) [Sabelfeld and Myers 2006] policies can be used to ensure component separation, but current techniques that enforce such policies either use costly runtime checks [Austin and Flanagan 2012; Austin et al. 2017; Tromer and Schuster 2016; Yang et al. 2016] or rely on advanced type systems in high level programming languages [Buiras et al. 2015a; Myers 1999a; Schoepe et al. 2014] that were not designed to develop embedded controllers. For example, Labeled Input Output (LIO) relies on Haskell's monads to soundly and effectively enforce IFC policies, which is well-suited for interacting with databases or web servers [Parker et al. 2019; Stefan et al. 2017].

But, direct application of LIO to embedded devices faces two major obstacles. First, LIO's policy enforcement relies on automatically generated runtime checks that would, upon failure, unpredictably crash the device. Second, Haskell's garbage collector and lazy evaluation might lead to memory leaks rendering automatic code extraction for devices with limited memory resources nearly impossible.

In this work, we develop Labeled Input-Outputs in F⋆ (LIO⋆), a variant of LIO that is implemented and verified in FStar (F⋆) [Swamy et al. 2016], and automatically extracted to efficient C code using the KreMLin (KreMLin) code generator [Protzenko et al. 2017]. Concretely, we use F⋆'s effects to encode IFC encapsulation in the spirit of LIO's monadic programming. This approach leads to three advantages: First, we lift policy enforcement from runtime checks to static proof obligations, using F⋆'s Dijkstra Monads [Maillard et al.

Authors' addresses: Jean-Joseph Marty, INRIA, IRISA, Rennes, France, jean-joseph.marty@inria.fr; Lucas Franceschino, INRIA, IRISA, Rennes, France, lucas.franceschino@inria.fr; Jean-Pierre Talpin, INRIA, IRISA, Rennes, France, jean-pierre.talpin@inria.fr; Niki Vazou, IMDEA Software Institute, Madrid, Spain, niki.vazou@imdea.org.

2019]. This leads, when possible (§ 2), to provably crash-free code. Second, the low-level code generated from KreMLin does not require a runtime library nor a garbage collector. Hence, it is ideally suitable for execution on embedded devices. Third, we use F⋆'s meta-programming support MetaStar (Meta⋆) [Martínez et al. 2019] to prove that the actual LIO⋆'s client applications enjoy non-interference. In short, we propose a methodology to generate both verified and runtime optimized IFC applications.

Our contributions are the following:

- We develop LIO⋆, an F⋆ library that enforces IFC policies (§ 3). LIO⋆ is statically verified by F⋆ which allows for both 1) more efficient implementations: since dynamic IFC checks can be statically proved, they become redundant at runtime (§ 3.3), and 2) low-level generated C code, that is automatically extracted using KreMLin.
- We design a mechanized meta-programming procedure that proves that client applications of LIO⋆ are non-interferent and apply it to two benchmarks and various case studies (§ 4). Concretely, we use F⋆'s meta-programming facilities (Meta⋆) to prove, for the first time, non-interference of the actual IFC client programs, in their source language, instead of idealized models of the language and programs. We report conclusions and some limitations of this endeavor.
- We benchmark LIO⋆ on three client applications (§ 5): 1) BUS⋆, that implements a bus system with IFC policies between communicating system components, 2) MMU⋆, that implements a software-defined memory management unit with policies on concurrent resources, and 3) DB⋆, that implements a database with explicit IFC policies. From these benchmarks, we conclude that extraction of C code under the IFC effect is possible and that the replacement of runtime checks with static proofs leads to cleaner code and up to 54% speedup (Table 7).

2 OVERVIEW

This section provides an overview of the interface of three IFC libraries we implemented, that are summarized in Figure 1. DLIO⋆, presented in § 2.1, provides IFC using runtime checks, which may unpredictably raise exceptions during execution, but requires zero static verification effort from the developers. SLIO⋆, presented in § 2.2, lifts IFC to the verification of static proof obligations, thus minimizing runtime overhead and failures. However, the developers are required to statically discharge all proof obligations. GLIO⋆, presented in § 2.3, both lifts all IFC checks as static proof obligations and marks all IFC privilege tracking information as computationally irrelevant to remove all IFC-related information from the runtime code, making it ideal for closed-loop execution but inconvenient when dynamic IFC information is necessary. We conclude that in applications where performance is critical, *e.g.,* embedded systems, the static effort required by GLIO⋆ might be worth the effort, but for mainstream software development SLIO⋆ provides an ideal trade-off among runtime checks and verification effort.

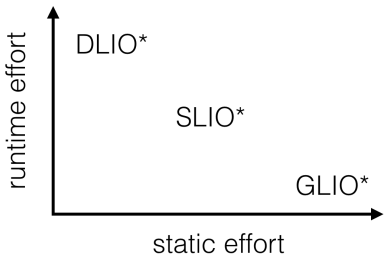


Fig. 1. The three versions of LIO*

2.1 DLIO⋆: naive translation of LIO into F⋆

We start with an overview of the interface of DLIO⋆, an IFC library that, like LIO, we uses a lattice of security labels to protect sensitive data.

Labels to protect values. We use security labels to express information flow control policies. As a trivial security label, consider an enumeration of three values: `type label = | Low | Medium | High`. To protect data, we assign them labels and ensure that data labeled as, *e.g.,* `Medium` can only be accessed by users with `Medium` or higher privileges.

The label system can be generalized to any lattice [Denning 1976] where the privileges hierarchy is defined by a partial order relation $\sqsubseteq$. The label interface is generalized as an F⋆ type class defined in § 3.1. Importantly, the `lattice` type class contains the least upper bound (`join`, $\sqcup$) of two labels, the label ordering (`canFlowTo`, $\sqsubseteq$), as well as proofs that the above methods form a partial order.

The IFC Effect. DLIO⋆ enforces IFC using runtime checks. Concretely, it provides two methods to label and unlabel data, that are defined in § 3.2 and have the interface below.

```
val label   : α → l → Ifc (labeled α)
val unlabel : labeled α → Ifc α
```

The method `label` takes a value `v` and a label `l` as inputs and returns a `labeled` value, *i.e.,* it wraps the value `v` with the label `l`. Dually, `unlabel` returns the value of its `labeled`, protected input. Both operations have an `Ifc` effect, which tracks the *current label* `cur`, that is, the accumulated privileges required to perform the current computation. Concretely, `unlabel lv` updates the current label by joining it with the label of `lv`. `label v l` only returns a labeled value when the current label can flow to `l` (`cur` $\sqsubseteq$ `l`). When it cannot, we are at risk of transmitting a value `v` of high privilege (*i.e.,* `cur`) to a level of lower security (*i.e.,* `l`). To prevent this violation, `label` fails with an exception.

Notations. To simplify the type signatures, we use α for types that support decidable equality α:`Type{hasEq` α`}` and l for instances of the `lattice` type class, a simplification permitted by using F⋆'s functors (as explained in § 3.1). Additionally, most trivial `ensures` and `requires` clauses, *e.g.,* `requires` λ `_` $\rightarrow \top$, are omitted from function signatures to improve readability.

Example. Consider below two functions that label and unlabel data below. On the left, `eqLabeled` takes two labeled data as input and unlabels them to compare their values. On the right, `checkLabeled` labels its input value and uses it to call `eqLabeled`.

```
let eqLabeled (v1 v2 :labeled α
  )                                  let checkLabeled (l:l) (i:α) (lv:labeled α)
              : Ifc bool                                  : Ifc bool
  let i1 = unlabel v1 in               let lv' = label i l in
  // cur := cur ⊔ labelOf v1           // exception if cur ⋢ l
  let i2 = unlabel v2 in i1 =          eqLabeled lv lv'
  i2                                   // cur := cur ⊔ labelOf lv
  // cur := cur ⊔ labelOf v2
```

Each time `eqLabeled` unlabels a value, the current label is joined with the label of the value, while when `checkLabeled` labels a value a runtime check is performed.

Extraction to C. DLIO⋆, as described so far, is equivalent to LIO where Haskell's `LIO` monad is replaced by F⋆'s `Ifc` effect. Wadler and Thiemann [2003] established that, in theory,

the transition from monads to effects is always possible. In practice, this transition from Haskell to F⋆ gives us the ability to extract low-level C code, using KreMLin [Protzenko et al. 2017] that automatically translates an F⋆ program to readable C code. For example, if our main program calls the function `checkLabeled (l,i)` with `label l` and `int32 i`, KreMLin will generate the following C code.

```
bool checkLabeled__int32_t(label l, int32_t i, labeled___int32_t lv) {
  label cur = getCurrent();
  if (!canFlow(cur, l)) fail("invalid labelling");
  labeled___int32_t lv_ = { .data = i, .tag = l };
  return eqLabeled__int32_t(lv, lv_);
}
```

That is, the C code directly accesses the current label `cur` and checks if it can flow to the argument label `l`. If so, it packs the input value `i` with the label `l` and uses it to call the C translation of the `eqLabeled` function. Otherwise, it fails with a runtime check. Below is the extraction of the `eqLabeled` function instantiated on integer labeled values.

```
bool eqLabeled(labeled___int32_t v1, labeled___int32_t v2) {
  labels c0 = getCurrent();
  labels c1 = join(c0, labelOf__int32_t(v1));
  setCurrent(c1);
  int32_t i1 = v1.data;
  labels c2 = getCurrent();
  labels c3 = join(c2, labelOf__int32_t(v2));
  setCurrent(c3);
  int32_t i2 = v2.data;
  return i1 == i2;
}
```

That is, for each call to `unlabel v`, the C translation 1/ gets the current label, 2/ joins it with the label of `v`, 3/ sets the current label to the joint label, and 4/ gets the value of `v`. Finally, the comparison of the two values is returned.

This translation from DLIO⋆ to low-level C brings us one step closer to our goal of generating an IFC library suitable to program embedded devices. But, we still face a big challenge: the generated code contains runtime checks (as seen in the above example) that are generated by calls to `label` and are not documented and thus unpredictable by DLIO⋆'s clients. These potential runtime failures make DLIO⋆ unsuitable for our goal, as hardware devices have critical sections that must not fail.

2.2 SLIO⋆: turning IFC runtime checks into static proof obligations

Next, we describe SLIO⋆, a static version of LIO⋆, where IFC checks are statically verified by F⋆, instead of being tested at runtime, as in DLIO⋆.

The crux of SLIO⋆ is that the IFC check performed by the `label` function is now lifted to a precondition of its `Ifc` effect [Maillard et al. 2019]. To call `label (v, l)`, the client needs to prove that the current label can flow to `l` as expressed by the `requires` statement `cur` $\sqsubseteq$ `l` below.

```
val label (v:α) (l:l) : Ifc (labeled α)
  (requires λ cur → cur ⊑ l)
  (ensures λ li x lf → labelOf x = l ∧ valueOf x == v ∧ lf = li)
```

```
val unlabel : (vl labeled α) : Ifc α
  (requires λ _ → ⊤)
  (ensures λ li x lf → lf = (li ⊔ labelOf vl) ∧ x == valueOf vl)
```

The `Ifc` effect that allows the expression of such a requirement is defined in § 3.3. To allow F* to discharge IFC requirements expressed as pre-conditions, functions with an `Ifc` effect must stipulate a post-condition, by mean of an `ensures` clause, that exactly captures their behaviors, *i.e.,* the returned value and how the current label is modified. For instance, as specified above, `label` leaves the current label unchanged, while `unlabel` joins its initial label `li` with the label of its input `vl`.

Propagation of Proof Obligations. The IFC requirement of `label` is propagated to all its clients. For example, the `checkLabeled` will not be verified unless we supply it with the `requires` below:

```
let checkLabeled (l:l) (i:α) (lv:labeled α): Ifc bool
  (requires λ l0 → l0 ⊑ l)
  (ensures λ li x lf → lf = li ⊔ l ⊔ labelOf lv) // definition as in 2.1

let eqLabeled (v1 v2 :labeled α) : Ifc bool
  (requires λ _ → ⊤)
  (ensures λ li x lf → lf = li ⊔ labelOf v1 ⊔ labelOf v2) // definition as in
  2.1
```

To enable verification of its clients, `checkLabeled` also specifies the modifications it performs to the current label by an `ensures` post-condition. In turn, the verification of `checkLabeled`'s post-condition is only possible by the above guarantee of `eqLabeled`, which itself verifies by the guarantee of `unlabel`.

Handing Dynamic Data with Path Sensitivity. Of course, it is not always possible to statically discharge proof obligations, especially when they depend on dynamic data. For example, the function `dynCheck` below uses a function `getDynamicLabel` that *dynamically* returns a label `l` (*e.g.,* from the console, a database or a random input). Then, it calls `checkLabeled` with that label.

```
let dynCheck (i:α) (lv:labeled α): Ifc bool =
  let l  = getDynamicLabel() in // e.g., getDynamicLabel () = intToLabel(
  getRandom())
  let lc = getCurrent() in
  if lc ⊑ l
    then checkLabeled l i lv
    else ...
```

Since it is impossible to statically prove that the current label can flow to the dynamic `l`, then the call to `checkLabeled` must be guarded by the required runtime check. Because verification in F* is path-sensitive, the call to `checkLabeled` will easily verify. The decision of what happens when the check fails is left to the user: if the user desires the code to be crash-free, then they should ensure the else-branch be properly covered, an alternative that was not existent in DLIO*.

Runtime-check free C code. Having paid the price of static IFC check propagation and verification, which in fact is highly aided by F⋆'s automation, we can enjoy C code without runtime checks. For example, the extracted C code for the `checkLabeled` function now simplifies to the below.

```
bool checkLabeled__int32_t(labels l, int32_t i, labeled___int32_t lv) {
  labeled___int32_t lv_ = { .data = i, .tag = l };
  return eqLabeled__int32_t(lv, lv_);
}
```

Now, `checkLabeled` simply packs the inputs to a labeled value and calls `eqLabeled`. The extracted code, compared to DLIO⋆, lacks all the runtime checks and exceptions derived from calls to `label`, leading to code amicable for unattended devices. But, code extracted from calls to the `unlabel` function remains unchanged. For instance, each call to `eqLabeled` is still updating the current label, which greatly slows down execution, especially in cases where it is actually unused.

2.3 GLIO⋆: ghosting the current label

The final variant of LIO⋆ is GLIO⋆ which, in addition to removing the dynamic IFC checks like SLIO⋆, also removes all updates to the current label before runtime. To do so, the current label is marked as computationally irrelevant, using F⋆'s `Ghost` module. In practice, this means that the current label is preserved at compile time, to verify static IFC proof obligations, but it is erased at runtime, thus the C extraced code is free from current state book-keeping.

The implementation is presented in § 3.4. and suffers from two inconveniences:

Inconvenience I: Lifting from and to ghost values. The `Ifc` effect keeps track of a ghost state that preserves an `erased` label. To pass this label to functions, *e.g.,* methods of the `lattice` type class that expect a label, we need to use F⋆'s `reveal` function that reveals the content of erased values within specifications. Dually, to turn labels into erased labels, *e.g.,* to compare them with the current label, we need to use F⋆'s `hide` function that erases values.

For example, the specification of `checkLabeled` gets polluted with `reveal` and `hide`, as below:

```
let checkLabeled (l:l) (i:α) (lv:labeled α): Ifc bool // definition in 2.1
  (requires λ li → reveal li ⊑ l)
  (ensures λ li x lo → lo = li ⊔ l ⊔ reveal (labelOf lv))
```

The semantics of the `requires` and `ensures` clauses remains the same, but now the argument label `l` exists at runtime, while the label arguments of the clauses are erased, thus conversions are required.

Inconvenience II: Dynamic Checks require explicit book-keeping. GLIO⋆ turns really inconvenient when static IFC checks involve dynamic data. For instance, consider that we want to replicate the function `dynCheck` from SLIO⋆, that calls `checkLabeled` with a label `l` obtained at runtime.

```
let dynCheck (i:α) (lv:labeled α): Ifc bool =
  let l  = getDynamicLabel () in
  let lc = ??? // was getCurrent () in 2.2
  if lc ⊑ l
```

```
    then checkLabeled l i lv
    else ...
```

In SLIO⋆, we used a runtime check to ensure that the current label can flow to `l`. This is not possible anymore, since the current label is erased at runtime. Now, what is a potential check we could perform to persuade F⋆ that calling `checkLabeled` with the dynamic `l` is valid?

We can always construct a label `lc` so that the `checkLabeled` call verifies, by explicitly passing around the current label. This essentially means that the clients need to manually and correctly, replicate all the current label accumulation that SLIO⋆ was automatically doing. Of course, this is error prone and not encouraged, but suggests that handling dynamic data is possible in GLIO⋆, though defeats its design. In short, unlike low-level applications, clients rely heavily on dynamic checks (as DB⋆ of § 5) will not benefit from GLIO⋆.

Current-label free C code. Though strenuous at times, GLIO⋆ comes with the huge advantage that the extracted C code is completely free from label information. For instance, below is the extracted C code from the functions `checkLabeled` and `eqLabeled`.

```
bool checkLabeled__int32_t(int32_t i, int32_t lv) {
  int32_t lv_ = i;
  return eqLabeled__int32_t(lv, lv_);
}

bool eqLabeled_int32_t(int32_t v1, int32_t v2) {
  int32_t i1 = v1;
  int32_t i2 = v2;
  return i1 == i2;
}
```

At runtime all labels are removed, thus labeled values are represented purely by their values. Thus, extraction to C erases not only all the current label book-keeping information, but also the data field selectors from labeled values. In short, the extracted code is light and amenable to low-level embedded systems, as supported by our benchmarks (§ 5). At the same time, IFC checks have been verified to statically hold by F⋆.

An additional benefit of GLIO⋆ is that, due to its lack of stateful updates, it is amenable to prove metatheorems. Concretely, in § 4 we define a metaprogramming procedure that encodes a noninterference lemma on clients of GLIO⋆. Intuitively, this procedure uses Meta⋆ to derive 1/ the "low view" of a GLIO⋆ function, *i.e.,* the view of an adversary with low privileges, and 2/ a lemma that encodes that low view is preserved by evaluation, and thus, as in Russo et al. [2008], encodes noninterference. We used this procedure on both the example functions `eqLabeled` and `checkLabeled`, as well as on two of our benchmarks (§ 5). In all these cases, F⋆ was able to automatically prove the noninterference, mechanically derived lemmata.

3 IMPLEMENTATION OF LIO⋆

This section presents the three versions of our IFC library, all of which express IFC policies using a verified label type class described at § 3.1. Concretely, we define
(§ 3.2) the dynamic DLIO⋆, where IFC policies are checked at runtime,
(§ 3.3) the static SLIO⋆, where IFC policies are lifted to compile time proof obligations, and

(§ 3.4) the ghost GLIO*, where the IFC label tracking is ghosted, *i.e.,* totally erased at runtime.

3.1 Labels as a type class

IFC policies are represented by a lattice [Denning 1976]. We encode the lattice of policies as an F* parametric type class that defines the lattice operations, constants, and properties. This encoding allows one to instantiate the type class to fit the policy of a domain-specific application, while ensuring all algebraic properties of the expected lattice operations are satisfied.

The "can flow to" operation ($\sqsubseteq$) defines the partial order relation between labels. The operations meet ($\sqcap$) and join ($\sqcup$) respectively define the greatest lower bound and the least upper bound of the two labels in this lattice. Finally, bottom ($\bot$) is the minimum lattice element. As in [Parker et al. 2019], the type class is refined with the required lattice properties:

- bottom is the smallest lattice value (`lawBot`),
- each label can flow to itself (`lawFlowReflexivity`),
- if two labels can flow to each other they are flow-equivalent (`lawAntisymmetry`),
- if three labels can flow in chain then the minimum can flow to the maximum (`lawTransitivity`),
- any label lower than a pair of labels can flow to the glb of these two labels (`lawMeet`), and
- the lub of a pair of labels can flow to any label accessible by this pair (`lawJoin`).

```
class lattice a = {
 ⊥: a;
 ⊑: a → a → Tot bool;
 ⊓: a → a → Tot a;
 ⊔: a → a → Tot a;

 lawBot: l:a → Lemma (⊥ ⊑ l);
 lawReflexivity : l:a → Lemma (l ⊑ l);
 lawAntisymmetry : x:a → y:a → Lemma ((x ⊑ y ∧ y ⊑ x) ⇒ x = y);
 lawTransitivity : x:a → y:a → z:a → Lemma ((x ⊑ y ∧ y ⊑ z) ⇒ x ⊑ z);
 lawMeet : z:a → x:a → y:a
   → Lemma (z = (x ⊓ y) ⇒  z ⊑ x ∧ z ⊑ y ∧ (∀ (l:a). l ⊑ x ∧ l ⊑ y ⇒ l ⊑ z)
  );
 lawJoin : z:a → x:a → y:a
   → Lemma (z = (x ⊔ y) ⇒ x ⊑ z ∧ y ⊑ z ∧ (∀ (l:a). x ⊑ l ∧ y ⊑ l ⇒ z ⊑ l))
  ;
}
```

To generate an IFC policy, one needs to instantiate the `lattice` type class with a concrete lattice. One such simple instance is, for example, the minimalistic classification of `Low` for public data, `Medium` for sensitive information, and `High` for private data:

```
type simple_lattice = | Low | Medium | High
```

The next step is to define the partial order `lcanFlow` using a comparison function `lt` between the three label values, and then the join and meet operations; as below:

```
let lt a b = match a, b with
  | Low, Medium → true
  | Medium, High → true
```

```
  | Low, High → true
  | _, _ → false
let lcanFlow a b = lt a b || a = b
let ljoin a b = if lt a b then b else a
let lmeet a b = if lt a b then a else b
```

The above functions define an instance of the `lattice` type class. All the lattice's laws, *e.g.,* `Lemma (⊥ ⊑ l)`, are automatically proved by F* using the above function definitions. Consequently, the instance does not need to provide any explicit proof terms, which are hence left as unit returning functions, *e.g.,* `λ _ → ()`.

```
instance SimpleLattice: lattice simple_lattice = {
  ⊑ = lcanFlow;
  ⊓ = lmeet;
  ⊔ = ljoin;
  ⊥ = Low;
  lawBot = (λ _ → ());
  lawReflexivity = (λ _ → ());
  lawAntisymmetry = (λ _ _ → ());
  lawTransitivity = (λ _ _ _ → ());
  lawMeet = (λ _ _ _ → ());
  lawJoin = (λ _ _ _ → ());
}
```

To simplify away the `lattice` type class constraints, our libraries behave as parametrized modules[1] over a lattice type. That is, in the rest we use the type l to refer to some `lattice` instance.

3.2 DLIO*: IFC with dynamic runtime checks

Next, we present DLIO*, an IFC library where security policies are dynamically checked with runtime checks. This library is basically LIO [Giffin et al. 2012] where the Haskell `LIO` monad is encoded as an F* effect.

Data protection in DLIO* is implemented by wrapping data with a security label to form a labeled value. To protect labeled values from policy-invalid access, an abstract public data type, called `labeled`, hides the actual data structure of the `private type` definition, called `labeledTCB`. The type `labeledTCB` is private and belongs to the library's Trusted Computing Base (TCB). As described in § 3.1, it uses l as an abstract data type that instantiates the `lattice` type class. To access the tag field of labeled values we define the operation `valueOf`. The data field is the goal of IFC protection, thus cannot be accessed.

```
private type labeledTCB α = { data: α; tag : l; }
type labeled α = labeledTCB α
let labelOf (vl:labeled α): l  = vl.tag
```

The IFC Effect. To securely access the value of labeled data we define the `Ifc` effect that accumulates the highest label required to perform that access. The `Ifc` effect is a state effect that carries a `context` and is defined below.

```
type context = {cur: l}
effect Ifc (a:Type) = IFC a (λ _ p → ∀ r. p r)
```

[1]Because parametrized modules *à la* OCaml are not available in F*, we trick, instead, F* module system.

The effect `Ifc` is parametric over the return type `a` and has trivially true pre- and post-conditions. To keep the description simple, we present the `context` to only be the current label, while our implementation also supports the LIO-style clearance optimization, defined as in [Giffin et al. 2012].

The `Ifc` effect comes with two primitive operations that set and get the current label.

```
private assume val setCurrent : l → Ifc unit
assume val getCurrent : unit → Ifc l
```

The setter function is required to manipulate the context of the `Ifc` effect, but can be used to break policy enforcement by arbitrarily setting a low current label. Thus, it is not exposed by our library, *i.e.,* is defined as `private`. Both of these functions are dropped during C extraction and are replaced with concrete C definitions. [2]

To run an `Ifc` computation, one should extract his program (either in OCaml or in C); F⋆ by itself is not intendeed at running code.

Data Unlabelling. To unlabel a labeled value `vl` the current label is raised to the label of `vl`, thus accumulating the privileges required to access `vl`.

```
let unlabel (vl:labeled α) : Ifc α =
  raise (labelOf vl);
  vl.data
```

Where the function `raise` sets the current label to the join of its argument and the old current label.

```
let raise (l:l) : Ifc unit =
  let c = getCurrent () in
  setCurrent (c ⊔ l)
```

Data Labelling. The function `label v l`, labels the input value `v` with the input label `l`, when the current label can flow into `l`, otherwise it fails with a runtime error.

```
let label (v:α) (l:l) : Ifc (labeled α) =
  let li = getCurrent ()  in
  if li ⊑ l
    then {data=v; tag=l}
    else fail "invalid label";
```

Where the function `fail` is a wrapper around C's failing function, assumed to be in the `Ifc` effect:

```
assume val fail : string → Ifc α
```

Addressing the "label creep problem". The "label creep problem" appears when an effectful computation, say `cmp`, requires access to sensitive data and as a result raises the current label too high. As in [Buiras et al. 2015b], to address this problem we define the `toLabeled` function that first computes the result of the sensitive computation `cmp`, then restores the current label to the original one, and finally returns the result of `cmp` labeled with the current label right after the `cmp` was computed.

[2]Defining these setter and getter functions directly using the canonical LowStar (Low⋆) way of dealing with C memory would litter our code with reasoning about the low-level memory model, a problem that will be addressed by F⋆'s layered effect, currently under development.

```
let toLabeled (cmp:b → Ifc a) (params:b) : Ifc (labeled a) =
  let c0 = getCurrent () in       (* save current label                    *)
  let v  = cmp params    in       (* run cmp with the given parameters     *)
  let c1 = getCurrent () in       (* get new current label                 *)
  let vl = label v c1    in       (* result with the new current label     *)
  setCurrent c0;                  (* restore the old current label, return
      *)
  vl
```

Since the result of `cmp` is labeled with the current label requires for `cmp`'s call and the current label is explicitly set to the original current labels, `toLabeled` addresses the label creep problem: the result of `cmp` is properly protected, while the current label remains unchanged.

In short, DLIO* is defined by naively implementing the original LIO implementation to F*. Yet, even this naive porting provides a great benefit. Taking advantage of KreMLin one can export DLIO* to high quality, low-level C code.

3.3 SLIO*: IFC with static proof obligations

Next, we present SLIO* where the IFC policy checks are turned into static proof obligations that are semi-automatically discharged by F*'s verification system. Intuitively, our goal is to lift the runtime check of `label` into a static assumption that prevents failure. To do so, we need to 1/ refine the `Ifc` effect with a context propagation weakest precondition and 2/ refine all `Ifc` functions with descriptive assertions, so that static verification precisely propagates the context information required to discharge `label`'s assumption.

The GTot effect to define specifications. In F*, specifications (namely *type refinements*, *ensures* and *requires* clauses) belong to the `GTot` effect, which stands for `Ghost` and `Total`. Their purpose is to serve static verification and they are erased at compile-time. This clear boundary of what is and what is not used at runtime, allows us to define functions that provably will not be called at runtime. For example, below we define the `valueOf` accessor

```
let valueOf (vl:labeled α) : GTot α = vl.data
```

A call to `valueOf` at runtime will simply destroy the whole purpose of our library, since it unconditionally accesses protected data. Yet, wrapped within the `GTot` effect, we rest assured that `valueOf` will only be used to define specifications and never leak the data of a labeled value at runtime.

Refinement of the Ifc effect. The `Ifc` effect is now indexed with a pre and post-conditions defined below using F*'s Dijkstra Monads [Ahman et al. 2017].

```
effect Ifc (a:Type) (pre:l → GTot Type0) (post:l → a → l → GTot Type0) =
  IFC a (λ (ci:context) (p:α → context → GTot Type0) → pre ci.cur ∧
          (∀ (v:a) (co:context).
             (pre ci.cur ∧ post ci.cur v co.cur) ⇒ p v co))
```

The `Ifc` effect definition is a standard state effect that consists of the return type `a` of the computation, the pre-condition `pre` on the the current label and the post-condition `post`. The pre- and post-conditions return F*'s proposition type `Type0` within the `GTot` effect. Using F*'s Dijkstra Monads we define, in a standard way, the weakest predicate transformer (WP) so that 1/ the pre-condition is valid with the initial context label `pre (c0.cur)` and that 2/ for any returned value `v` and final context `co` that satisfy the pre- and post-condition, then the post-condition `p v co` also holds.

With this indexing, the `Ifc` functions can use precise `ensure` predicates to describe their behavior. For example, with refine the `getCurrent` and `setCurrent` operations as follows.

```
assume val getCurrent : unit → Ifc (l)
  (ensures λ li x lo → li = lo ∧  x = lo)

private assume val setCurrent : (l:l) → Ifc (unit)
  (ensures λ li _ lo → lo = l)
```

The function `getCurrent` ensures that the current label is preserved (*i.e.,* `li = lo`) and returned (*i.e.,* `x = lo`). The function `setCurrent l` ensures that the current label is set to `l` (*i.e.,* `lo = l`). F* also needs requires clauses, where when omitted the trivial `(requires λ _ → ⊤)` is implied.

Data Unlabelling. The definition of data unlabelling, *i.e.,* `unlabel`, and label raising, *i.e.,* `raise`, are exactly the same as in DLIO*. But now, both functions come with specifications that precisely capture their behavior.

```
let unlabel (vl:labeled α): Ifc (α)
  (ensures λ li x lo → lo = li ⊔ (labelOf vl) ∧ x == valueOf vl)

let raise (l:l) : Ifc unit
  (ensures λ li x lo → lo = li ⊔ l)
```

The ensure clause of `unlabel` specifies that the output label is the join on the input and the label of the argument, while the returned value is the value of the argument. The verification of this definition is only possible when the `raise` function also comes with a precise ensure clause that, as above, states that the current label is joint with the function's argument.

Data labelling. The crux of the SLIO* library is that the definition of `label` is now changed to be crash-freedom free. Now `label v l` simply labels the value `v` with `l`, while its precondition ensures that no information is leaked.

```
let label (v:α) (l:l) : Ifc (labeled α)
  (requires λ li → li ⊑ l)
  (ensures λ li x lo → (labelOf x) = l ∧ (valueOf x) == v ∧ lo = li)
= {data=v; tag=l}
```

IFC is captured by the requirement that the current label should flow to the argument label `l`, while the `ensure` clause precisely captures the function's behavior of `label` to aid static verification of `label`'s clients.

Addressing the "label creep problem". The `toLabeled` function that addressed the label creep problem is the same as before. Yet, to pass static verification yet, its type signature requires a slight modification. Concretely, we use F*'s argument metaprogramming $ key to define the type of `toLabeled` as follows:

```
let toLabeled #a #b #pre #post
  ($cmp:b→Ifc a (requires pre) (ensures post))
  (params:b)
  : Ifc (labeled a)
  (requires λ li → pre li)
  (ensures λ li x lo → li = lo ∧ post li (valueOf x) (labelOf x))
```

The arguments `a b pre post` are implicit, thus marked with `#`. The input computation `cmp` is infixed with the `$` key that disable subtyping, inferring the implicit `pre` and `post` arguments based on `cmp`'s pre- and post-conditions. Then, the precondition is propagated as the `toLabeled`'s precondition to allow the call to `cmp` statically verify. The postcondition of `toLabeled` ensures that the current label is not modified. With this type signature, the definition of `toLabeled` is left the same as before and now the call of `cmp` is statically verified, since its precondition is propagated as a precondition to `toLabeled`.

In short, SLIO* has lifted the IFC runtime check of `label` to a static assumption. This assumption propagates to direct and indirect clients of `label`. To discharge these assumptions, all SLIO* functions come with precise postconditions. Then verification proceeds using the weakest preconditions of F* as specified in the definition of the `Ifc` effect. This way, SLIO* clients are extracted to efficient C code that is runtime check free.

3.4 GLIO*: label tracking becomes ghosted

Finally, we present GLIO* where all label tracking information is explicitly marked as computational irrelevant – using F*'s ghost mechanism – and are thus removed from runtime.

Computational Irreleval Values. To encode computationally irrelevant values we use three functions below from F*'s `Ghost` module. Erased values are decorated with the erasable attribute `E`. The `reveal` function allows to access labeled values within the `GTot` effect, hence at compile-time only, whereas the `hide` function allows to erase them.

```
module G
  type erased (a:Type) = | E of a
  val reveal : #a:Type → erased a → GTot a
  val hide   : #a:Type → a → Tot (erased a)
```

Ghosted Labeled Data. The label field of a labeled value is now marked as erased: it is used to check policy enforcement at verification time and to make it explicitly unavailable at runtime. The definition and accessors of the labeled type are as follows:

```
private type labeledTCB α = { data:α;  tag:G.erased label; }
type labeled α    = labeledTCB α (* public version *)
let valueOf (vl:labeled α) : GTot (α) = vl.data
let labelOf (vl:labeled α) : (G.erased l) = vl.tag
```

The public version of the labeled type is, as before, a wrapper around `labeledTCB`, which now contains an erased label. The value accessor remains unchanged, and can access the data at compile-time under the `GTot` effect. Importantly, the label accessor now returns an erased label, which pass be arbitrarily passed around, but can only be revealed in specifications, *i.e.,* under the `GTot` effect.

Ghosted IFC Effect. We further erase the current label field of the context.

```
type context = { cur:G.erased l;}
```

In our implementation, the context also contains a predicate over labels that we use to encode clearance-style optimizations. Yet, for simplicity here we omit this field.

We adjust the `Ifc` effect of SLIO* to account for the fact that now the current label is erased. Concretely, each access to the current label needs to also reveal its value, which is

possible, since specifications live in the GTot effect. We declare gc as the function that first accesses the current label and use it to define the Ifc effect as follows:

```
let gc (c:context): GTot l = G.reveal c.cur
effect Ifc (a:Type) (pre:l → GTot Type0) (post:l → a → l → GTot Type0) =
  IFC a (λ (ci:context) (p:α → context → GTot Type0) → pre (gc ci) ∧
          (∀ (v:a) (co:context).
             (pre (gc ci) ∧ post (gc ci) v (gc co)) ⇒ p v co))
```

Compared to the Ifc definition on SLIO⋆, we simply switched c.cur to gc c.

The getter and setter functions of the ghost Ifc effect will also not be used at runtime. So, instead of wrapping C code (as in SLIO⋆) now use the private Ifc functions IFC?.get and IFC?.put that are provided by F⋆ as part of the effect definition.

```
let getCurrent (_:unit) : Ifc (G.erased l) =
  let cc = IFC?.get () in cc.cur
private let setCurrent (l:G.erased l) : Ifc (unit) =
  IFC?.put {cur = c}
```

The getter function getCurrent returns the erased current label, which can be revealed only at compile-time. The setter function setCurrent could still be used by clients to violate IFC policies, so remains part of the libraries TCB and thus, is marked as private.

The same holds for the toLabeled function, and in general each function that does not perform lattice operations (*e.g.,* ⊑, ⊔) on erased labels. When lattice operations are performed, like the functions raise, label, and unlabel below, careful conversion is required between erased and actual labels.

Raising the Current Labels. The function raise that is joining the current label on with its argument now takes as input an erased label. This edit is required, since in practice labels are generated after operations with the current label. Since the current label is not erased and cannot be revealed by clients without lifting the whole client as ghost, the label arguments to functions are turned into erased labels. Dually, it is always feasible to generate erased labels within the Ifc effect. That is because hide, that erases labels, is defined in the Tot effect, which is automatically lifted to Ifc.

The definition and specification of raise is the following:

```
let raise (l:G.erased l) : Ifc unit
  (ensures λ li x lo →  lo = li ⊔ G.reveal l) =
  let li = getCurrent () in
  setCurrent (G.hide (G.reveal li ⊔ G.reveal l))
```

The ensure clause simply reveals the argument label l, since it is in the GTot effect. To join the current labels li with the argument label l both need to be revealed, producing the GTot effect. This effect though is encapsulated by the hide operation that turns the result on the join backed to erased.

Data Labelling and Unlabelling. The definitions of data labelling and unlabelling do not use any lattice specific operations, thus remain unchanged. Yet, their specifications require the addition of the reveal function each time erased labels are passed to lattice methods or compared. Concretely, the specifications of unlabel and label turn to the following:

```
let unlabel (vl:labeled α): Ifc (α)
  (ensures λ li x lo → lo = li ⊔ G.reveal (labelOf vl) ∧ x == valueOf vl)
```

```
let label (v:α) (l:G.erased l) : Ifc (labeled α)
  (requires λ li → li ⊑ G.reveal l)
  (ensures λ li x lo → G.reveal (labelOf x) = G.reveal l ∧ valueOf x == v ∧ lo =
   li)
```

The only difference compared to the respective specifications of SLIO⋆ is that `G.reveal` is properly inserted on erased labels, before the lattice operations or equality is performed. This addition is an inconvenient, by type-directed process.

Retrieving The Ghost Current Label. The fact that in GLIO⋆ the run-time representation of the current label is erased does not imply a loss of expressivity, since with great effort, one could emulate the current label at run time.

In fact, GLIO⋆ is as expressive as SLIO⋆, since SLIO⋆ can be implemented as an external definition on the top of GLIO⋆. Such a definition will take the shape of a state monad, carrying —at runtime— a label; it would provide wrappers for the `label`, `unlabel`, `raise` and `toLabeled` operations, acting as user defined proxies to GLIO⋆'s own operations. These definitions would be written so that the state monad would maintain a specific F⋆ invariant: the runtime label stays equal to GLIO⋆'s erased label.

In practice, a GLIO⋆ user would selectively choose to pass around a runtime label that is provable to be equal to GLIO⋆'s erased label. Obviously, proving this label coherency property has the non-negligable human-time cost of writing proofs.

In short, GLIO⋆ is using the ghost module of F⋆ to mark the current label and the label inside labeled values as computation irrelevant, and thus erase it at runtime. The co-existence of erased and non erased labels makes writing specifications an inconvenient process, where reveal and hide annotations have to explicitly be provided. On return, as overviewed in § 2 and benchmarked in § 5 clients of GLIO⋆ translate to cleaner and more efficient, low-level C code.

4 META-METATHEORY: PROOFS OF NON INTERFERENCE

In this section we make use of F⋆'s metaprogramming facilities (Meta⋆ [Martínez et al. 2019]) to define a procedure, concretely the metaprogram `genNILemma`, that takes as input a toplevel name of a GLIO⋆ client, say `f`, and generates a lemma statement that `f` is noninterferent. We applied this procedure to two benchmarks from § 5 and the examples from §2. In all these cases F⋆ automatically proved the correctness of the mechanically derived, noninterference lemma.

4.1 Statement of Non-interference

Consider a GLIO⋆ function `f` that given an argument of type α returns an `Ifc` computation with a precondition `pre`, postcondition `post`, and return value of type `b`.

```
val f : α → Ifc b (requires pre) (ensures post)
```

We express noninterference of `f` using the low view preservation of Russo et al. [2008]. That is, `f` is noninterferent when its low view is preserved by evaluation. This low view intuitively represents the view of an adversary with low privileges. More concretely, the low view on a level `l:`l is defined by an ϵ_1 function that forgets (*i.e.,* replaces by a "hole") all the data that are protected by a label higher than `l`. The evaluation of the `Ifc` function `f` with an argument `x` on the initial context `c` is denoted as $\downarrow^{\mathtt{c}}\mathtt{f(x)}$. With these notations, we express the noninterference lemma, *i.e.,* preservation of erasure by evaluation, as follows:

Lemma: fNI $\equiv \forall(\texttt{l}{:}l)\ (\texttt{x}{:}a)\ (\texttt{c}{:}\texttt{context}\ \{\texttt{pre c}\}).\ \epsilon_1(\downarrow^{\texttt{c}}\texttt{f}(\texttt{x})) = \epsilon_1(\downarrow^{\texttt{c}}(\epsilon_1(f))(\texttt{x}))$

The above fNI lemma states that the evaluation of `f` (*i.e.,* $\downarrow^{\texttt{c}}\texttt{f}(\texttt{x})$ on the left hand side) and the evaluation of its erasure (*i.e.,* $\downarrow^{\texttt{c}}(\epsilon_1(f))(\texttt{x})$ on the right hand side[3]) cannot be distinguished after erasure, for any erasure level `l`, input `x`, and initial context `c` that satisfies `f`'s precondition.

Our goal is to define a metaprogram that takes as input the binder of a function `'f` and encodes the noninterference lemma for `f`. To do so, we need to encode evaluation and erasure in F*.

Encoding of evaluation. In F* evaluation of effectful computations is encoded by Filinski [1994]'s reification that changes effectfull calls from implicit monadic style to explicit passing style. That is, the evaluation $\downarrow^{\texttt{c}}\texttt{f}(\texttt{x})$ in the noninterference lemma fNI is encoded in F* as `reify (f x) c`.

Encoding of erasure. In fNI erasure is used to erase both 1/ the results produced after evaluation and 2/ the function `f` itself. For the first case we define the `eraseCtx` function below.

```
let eraseCtx [|hasEraser α|] (l:l) (x:α,c:context) :
    α
  if c.curr ⊑ l then erase x else •
```

```
class hasEraser α
  { erase :: l → α → α }
```

The function `eraseCtx` on the left, takes as input an erasure level `l` and the reification pair `(x,c)`. If the label of the input context can flow to `l` (*i.e.,* `c.curr` $\sqsubseteq$ `l`), then the value `x` is erased, using the erasure method `erase` we define on § 4.2. Otherwise, a 'hole' (also defined on § 4.2) is returned.

When erasure is used on the function `f` itself, *i.e.,* the appearance $\epsilon_1(f)$ in the fNI lemma, it returns a different function, named `f_erased l`, that our metaprogramming procedure systematically generates. Intuitively, the top level function `f` with type `t`, generates a new top level function named `f_erased` with type $l \rightarrow$ `t`, *i.e.,* the erasure level is explicitly passed, where all sensitive data protected with labels above `l` are erased. In § 4.3 we describe the generation of erased functions.

NI Lemma Generation. To generate the noninterference lemma we call the metafunction `genNILemma`.

```
val genNILemma : Term → Tac unit
```

`genNILemma` takes as input the binder representation of a top level GLIO* function, *e.g.,* `'f`, and generates a top level declaration that defines the noninterference lemma of `f`, named as `f_NI`. `genNILemma` takes as input a `Term`, that is the meta-representation of an F*'s AST defined in Meta*, and has the `Tac` effect that allows inspection of the representation of values as well as function definitions, similar to TemplateHaskell's `Q` monad [Sheard and Jones 2002]. It returns a `unit` value, since the top level `f_NI` lemma generation is an effect.

The function `genNILemma` is a meta-program that generates one or multiple top level definitions, executed by the means of the instruction `splice`. For example, `splice [] (genNILemma 'f)` generates the non-interference lemma below. The `[]` part of `splice`, required for scoping, is in the rest left empty for simplicity.

[3]Note that the argument `x` does not need to be erased. Evaluation under erasure (*i.e.,* $\downarrow^{\texttt{c}}(\epsilon_1(f))(\texttt{x})$) ensures that each usage of the argument `x` on the erased `f` will be actually erased.

```
%splice[](genNILemma `f) \\ generates the below lemma
let f_NI (l:l) (x:a) (c:ctx{pre c})
 : Lemma (eraseCtx l (reify (f x) c) == eraseCtx l (reify (f_erased l x) c))
 = ()
```

The lemma generation metaprogram `genNILemma` supports functions with many arguments. Concretely, it inspects the type of the input binder, and if it has `n` arguments, it creates the lemma arguments `x1..xn` with the proper types and uses them to call the original and erased functions.

For example, for the definitions of `eqLabeled` and `checkLabeled` of the overview (§ 2), the respective generated lemmata are shown below.

```
%spice[](genNILemma `eqLabeled_NI `checkLabeled) \\ generates the below lemmata
let eqLabeled_NI (l:l) (x1:labeled α) (x2:labeled α) (c:context)
 : Lemma (eraseCtx l (reify (eqLabeled x1 x2) c) ==
          eraseCtx l (reify (eqLabeled_erased l x1 x2) c))
 = ()

let checkLabeled_NI (l:l) (x1:l) (x2:α) (x3:labeled α) (c:context {l.curr ⊑ x1})
 : Lemma (eraseCtx l (reify (checkLabeled x1 x2 x3) c) ==
          eraseCtx l (reify (checkLabeled_erased l x1 x2 x3) c))
 = ()
```

The erased functions `checkLabeled_erased` and `eqLabeled_erased` are defined at top level by `genNILemma` and are presented in § 4.3.

F* was able to automatically prove both the above lemmata, *i.e.,* with the trivial unit body definition, thus showing that neither of these functions interferes. Further, using `genNILemma` we proved two out of our three benchmarks (BUS* and MMU*) are noninterferent, while the third (DB*) imposes technical limitations, we discuss in § 4.4.

4.2 Erasure of Values

Here we describe the definition of the `erase l x` methods from § 4.1 that is erasing, *i.e.,* replacing by a "hole", all data inside the value `x` that are protected by a label higher than `x`.

The hole is encoded in F* as •, *i.e.,* an axiomatized polymorphic value defined below.

```
assume val •: (α: Type) → α
```

Thus • has no content and can used to replace sensitive data of any type.

Erasing a labeled value. To erase a labeled value, we define the function `eraseLabeled` below

```
let eraseLabeled [|hasEraser α|] (l:l) (x:labeled α) : labeled α =
  if x.tag ⊑ l
    then { data=erase l (x.val); tag=x.tag }
    else { data=•; tag=x.tag }
```

Erasure of the labeled value `x` checks if the `tag` of `x` can flow to the erasure level `l`. If it can, it recursively erases the data, otherwise it replaces the `x`'s data with a •. At each case, the `tag` field remains untouched. There are two things to note here: First, our implementation forcefully violates the `labeled` privacy, specified in the GLIO* implementation. Second, all the erased labeled (*i.e.,* in the tag field) are revealed. Both of these are implemented using a trick on F* module system to emulate lacking OCaml-like module functors.

We use the `eraseLabeled` definition to define the `hasEraser` instance on the `labeled` type.

```
let labeledHasEraser [|hasEraser α|]: hasEraser (labeled α) = {
  erase l x = eraseLabeled l x
}
```

For the rest types the `hasEraser` instance definition acts as a homomorphism. For *primitive types* (like $\mathbb{B}$, $\mathbb{Z}$ or *unit*), we defined the `hasEraser` instance to be the identity. For *inductive data types* were defined a metaprogram that using generic programming techniques automatically defines the `hasEraser` instances. For example, for lists, our metaprogram follows the list structure to mechanically generate the `eraseList` below.

```
let rec eraseList [|hasEraser α|] (l:l) (x:list α) : list α =
  match x with
    | Cons hd tl → Cons (erase l hd) (eraseList l tl)
    | Nil → Nil
```

The `eraseList` is used, like in `eraseLabel`, to define the list `hasEraser` instance.

In short, our metaprogram mechanically generates `hasEraser` instances for inductive types, while generation of instances for other types (*e.g.,* arrow types or higher kinded types), when required, is left to the user.

4.3 Erasure of Functions

Function erasure is a metaprogram, `eraseF :: Term → Tac unit` that given a GLIO* top level binder, say `'f`, generates the top level definition `f_erased` as follows. The first argument of `f_erased` is a label `(lErase : l)`. We define erased function parametric on erasure level to avoid definition of multiple functions for different erasure levels. Then, the specifications are left unmodified, while the body definition of `f_erased` follows the AST of `f` where 1/ each labeled subterm is erased and 2/ each function binder is replaced by its erased version, as explained below.

1. Erasure of Labeled Subterms. If `e` is a labeled subterm in the function definition, it is replaced by `eraseLabeled lErase e`, where `lErase` is the erasure level argument introduced to `f_erased`. To check if `e` is a labeled term, `eraseF` defined in the `Tac` effect, simply type checks the expression `eraseLabeled lErase e`. Thus, for each subterm `e`, if `eraseLabeled lErase e` type checks it always replaces `e`, leading to some benign extra checks. Of course, the subterms `e` are open, *i.e.,* have unbounded variables, thus during AST traversal we keep an environment of the introduced variables and use it to type check subterms.

2. Erasure of Function Binders. If `g e1 ... en` is a subterm of the original function, with `g` being a function symbol, then `g` also needs to be erased. By default, we replace the function call with `g_erased lErase e1 ... en` and recursively call `eraseF` on `'g` to define the erased declaration of `g`, if it is not already defined. We keep a list of function binders that do not need to be erased and calls to such functions remain untouched. First, we do not erase the functions defined in the GLIO* since their erasure is provably an identity, as captured by their specifications. Second, we do not erase functions imported from F* standard libraries, *e.g.,* map, `=`, `+`, ...

Axiomatization of Contamination. The decision not to erase primitive F* functions is made to avoid code expansion, and highlights a problem we call *contamination*. Contamination captures the • propagation from arguments to results, *e.g.,* `• = 42` should be equal to •. In general, if a function `g` *consumes* it's ith argument, then the call of `g` with a • on

the ith position, should be equal to $\bullet$. Our implementation axiomatizes contamination for primitive functions, *e.g.,* below we provide the contamination axioms for `=` and `+` on their first argument.

```
let contaminationEq1   n : Lemma (• = n == •) = admit ()
let contaminationPlus1 n : Lemma (• + n == •) = admit ()
```

Note that contamination axioms on the second argument are not required, since the SMT solver will derive them by commutativity. Contamination axioms are not required for most functions from the standard library, *e.g.,* `map` and `fold` that can be normalized. To aid contamination axiom generation and limit errors we mechanised axiom generation by the implementation of a metaprogram that given the function name and contamination position derives the proper contamination axiom.

Examples of Function Erasure. To illustrate the function erasure process, reconsider the `checkLabeled` and `eqLabeled` functions from the overview (§ 2.1). Their erased version produced by calling `eraseF` via `splice` are presented below.

```
%splice[add_erased, checkLabeled_erased] (eraseF (lookup (‘checkLabeled)))

let eqLabeled_erased (lErase: l) (v1 v2:labeled α)
  : Ifc bool =
  let i1 = unlabel (eraseLabeled lErase v1) in
  let i2 = unlabel (eraseLabeled lErase v2) in
  i1 = i2

let checkLabeled_erased (lErase: l) (l:l) (i:α) (lv:labeled α): Ifc bool
  (requires λ li → reveal li ⊑ l)
  (ensures λ li x lo → lo = li ⊔ l ⊔ reveal (labelOf lv))
  let lv' = eraseLabeled lErase (label i l) in
  eqLabeled_erased lErase (eraseLabeled lErase lv) (eraseLabeled lErase lv')
```

From the above examples we note that the functions specifications remain unchanged, though now `reveal` and `hide` are now identities, as discussed in § 4.2. The erasure level argument `lErase` is passed around to prevent multiple definitions of erased functions on different erasure levels. The functions `label`, `unlabel` are not erased because the belong in GLIO* and `=` is not erased because it is a primitive F* function (it is though axiomatized for contamination). The function `eqLabeled` is erased. If fact, in the above `splice` the original call of `checkLabeled` to `eqLabeled` triggered the declaration of `eqLabeled_erased`. Finally, we observe a redundancy on `eraseLabeled` wraps. The binder `lv'` is wrapped both at its definition and call site. Such redundancies can be syntactically detected and removed, thought they do not affect our goal that is static proof of noninterference.

4.4 Discussion & Limitations

The implementation of the metaprogramming procedure for noninterference lemma extraction `genNILemma` is about 800 Lines of Code (without spaces, with comments). We abstracted the GLIO* dependencies of the implementation in such a way that `genNILemma` can be easily

adapted by other IFC libraries. Due to continuing modifications of the API of Meta*, our implementation is build against a specific F* built[4].

Our approach has three main limitations.

Immaturity of Meta*. Our implementation is a heavy client of Meta* which is quite immature. Published in 2019, Meta* has still various limitations, most importantly the inability of full AST inspection (*e.g.,* arbitrary effect weakest preconditions cannot get inspected). More, Meta* is rapidly changing, which makes development and use of our library inconvenient. Yet, exactly due to these changes, we believe that is the near future all the current limitation will get addressed.

Reifiable Requirement. The main limitation of `genNILemma` is that, to state noninterference of an `Ifc` function `f` we use reification of `f`, thus `f` should live in an effect that is reifiable to a total computation. For example, potentially diverging expressions cannot be handled `genNILemma`, which is expected since most noninterference proofs are termination sensitive [Parker et al. 2019; Stefan et al. 2017]. This limitation is the reason why we did not apply the DB* benchmark (from § 5) to `genNILemma`. Our benchmark needs access to a database, thus has an effect that is not directly reifiable to total.

There is a way to address this limitation, which is to *assume* the nonreifiable effect using a specification. That is, to give static semantics by means of a dependent-type, but no dynamic semantics, *i.e.,* no implementation. In such a case, one is able to conduct proofs, but unable to normalize programs (as some primitives have no implementation), which is necessary to properly conduct the proofs on generated theorems. In the near future, we plan to follow this approach and apply `genNILemma` to our DB* benchmark.

Specificity. The generated noninterference proofs are very specific. Each proof is developed for exactly one client of GLIO*. But, our metaprogram is defined for every GLIO* client constrained by the above limitations.

Our decision to specifically target clients of GLIO* instead of proving the correctness of the GLIO* library stems from the fact that our goal is to verify real executable code. This is difficult, since real code comes with various verification unfriendly constructs. Thus, instead of following the route of [Parker et al. 2019; Stefan et al. 2017], that is to design a model of LIO and prove it correct, we restricted ourselves to per-client proof construction. This approach comes with another huge benefit. Both library noninterference proofs of [Parker et al. 2019; Stefan et al. 2017] impose limitation on the programs that do not interfere, concretely, those programs should be terminating and "safe" which intuitively means that they should not access the library's TCB. In our proof such assumptions do not exist. Instead, if `genNILemma` is called on programs that violate the safety assumption, F* would not be able to prove correct the derived noninterference lemma.

In short, our metaprogramming approach targets clients of the real GLIO* library instead of proving the correctness of a verification friendly model of the library. The noninterference lemma is mechanically derived per client, by a simple call to `genNILemma` and the proof is automated by F*.

5 BENCHMARKS

In this section we present three benchmarks: database in F* (DB*) in § 5.1, BUSStar (BUS*) in § 5.2, and Memory Management Unit in F* (MMU*) in § 5.3 and explore how their

[4]We use a patched version of the F* commit `0362a90a83bea851fa5e720637f1cb9d3dfe97bc`, for more detail see the Nix expression in the implementation sources.

implementation on the three versions of our library (DLIO⋆, SLIO⋆, and GLIO⋆) affects the size of the development F⋆ code, the size of the extracted C code, as well as the runtime performance of the benchmarks.

5.1 The DB⋆ case study

Our first case study is DB⋆, which is inspired by the λChair conference management system of [Stefan et al. 2017]. The goal of DB⋆ is to perform the transactions of a conference selection process: submission, assignment, review, selection, results, while maintaining anonymity of reviewers and respecting conflicts of interest. We ported λChair examples to all three versions of LIO⋆.

The example below is using GLIO⋆ to implement the scenario where the user `Charles` reviews all the papers submitted by the user `Mary`. Concretely, the call `fetchPapers_for Mary` brings all the `labeled` papers of `Mary` and the call `map add_review_from_Charles` adds a review from `Charles` for each of the labeled papers.

```
let example (_:unit) : Ifc unit
  (requires λ _ → ⊤)
  (ensures λ li _ lf → li == lf) =
  let l = fetchPapers_for Mary in  (* get all Mary's labeled papers
  *)
  (* the current label does not grow because the papers are not opened
  *)
  map add_review_from_Charles l    (* Charles will add his review to each paper
  *)
  (* the map executes toLabeled for each paper so the context is still preserve
  *)
```

The `ensures` clause above requires that the current label is preserved. Since `fetchPapers_for` returns labeled papers it does not raise the context, but for reviewing higher privileges are required. Locally raising the current label can only be achieved by the `toLabeled` function.

When we use GLIO⋆, where labels of `labeled` values are ghosted we face a problem. We are required to locally raise the current label using the dynamic labels that protect `Mary`'s papers, which is impossible. To address this problem, in this DB⋆ benchmark, we boxed `labeled` values with a label, that exists at runtime and *soundly* approximates the ghost label of the `labeled` value. We define the `box` type as follows:

```
type box a = {
  data: labeled α;
  tag: (l: l{gc (labelOf data) ⊑ l})
}

let unbox (bv:box α) : Ifc α
(ensures λ li x lf →  (gc lf) = ((gc li) ⊔ bv.tag) ∧ x == valueOf (bv.data)) =
  unlabel bv.data
```

The refinement type in the `tag` of the `box` ensures that the `box`'s tag can be safely used at runtime when the protection label is required. In the GLIO⋆ implementation of DB⋆ we store `box` values in the database – to be able to access the labels at runtime – while the DLIO⋆ and SLIO⋆ implementations store `labeled` values.

In Table 5 we observe that the lines of F* program code (LoP) of the implementation increased from 166 in DLIO* (which is very similar to the original LIO implementation) to 265 in SLIO* and to 269 in GLIO*. This is a big increase on Lines of F* (LoF) which depicts that the verification effort required was strenuous. The reason for that is that this specific example was developed to present the expressiveness power of LIO and it is very heavy on dynamic IFC checks. In Table 7 we observe that this verification effort does not pay off at runtime speedups which are only 9.1% and 12.6% for SLIO* and GLIO*, respectively.

From this benchmark we conclude that all the libraries are equally expressive, but clients with heavy runtime checks are advised to use the dynamic DLIO* version.

5.2 The BUS* case study

Our second benchmark is an BUS* Application Programming Interface (API) that uses IFC to ensure component separation. Figure 2 gives flowchart of the BUS*'s algorithm which is triggered any time a packet is available for transport.

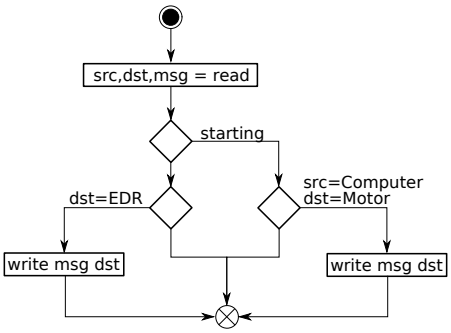


Fig. 2. Flowchart of `readBUS`

We enforce component separation using LIO* as follows. For each `actor`, *i.e.,* a system component, we define a security label (*i.e.,* `type component = l`). The current label of the `Ifc` effect keeps track of the accumulated BUS label. Each time an `actor` reads from the BUS, the current label is raised with `actor`. Dually, each time an `actor` writes to the BUS, it is required that the current label can flow to the `actor`. This behavior is similar to LIO's `unlabel` and `label` operations. Thus, to enforce component separation we simply wrap the BUS read and write primitives to the `Ifc` effect via `unlabel` and `label` operations. Below we provide the wrappers as implemented in GLIO*.

```
let writeBUS (actor:component) (data:byte) : Ifc unit
  (requires λ li → li ⊑ actor) (ensures λ li x lo → li = lo) =
  write actor (label data (G.hide actor))

let readBUS (actor:component): Ifc byte
  (ensures λ li x lo → lo = li ⊔ actor) =
  unlabel (read actor)
```

BUS execution:	*BUS label*	EDR *label*	Computer *label*	Motor *label*
	∅	E	C	M
let a = read Computer in	C	E	C	M
(write a EDR);	C	E⊔C	C	M
let a = read Motor in	C⊔M	E⊔C	C	M
(write a EDR);	C⊔M	E⊔M⊔C	C	M

Fig. 3. Event Data Record Example using in BUS*

Figure 3 presents an Event Data Recorder (EDR) example, where BUS⋆ controls the transport of data from the computer, the motor controller and the EDR. Using BUS⋆ to implement an EDR requires two IFC policies. First, when the system starts, the computer is allowed to communicate with the engine. Second, the computer and the engine cannot exchange any information but can only communicate with the EDR to register event data.

We implemented this EDR example in DLIO⋆, SLIO⋆ and GLIO⋆. In Table 5 we observe that our implementation is 94 LoP in all versions, since the client did not require any static specifications. The lines of extracted C code Lines of Code (LoC) greatly reduced from 100 LoC in DLIO⋆, to 92 LoC in SLIO⋆, and to 68 LoC in GLIO⋆. This reduction expresses that the few runtime checks imposed by our example were trivially verified, rendering runtime label book tracking useless. In Table 7 we observe that the running time of the example is in accordance with the extracted C code size and gives 21.4% speedup for SLIO⋆ and 54.6% for GLIO⋆.

This benchmark illustrates that for applications that are not heavy on IFC checks and do not depend on dynamic labels (in contrast to § 5.1), moving from DLIO⋆ to GLIO⋆ gives significant runtime speedup with trivial verification cost effort.

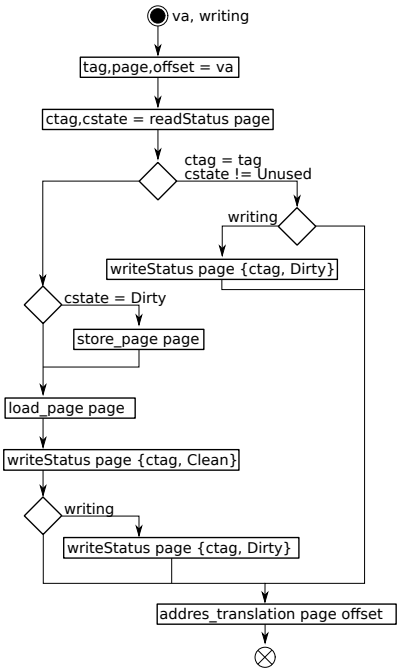


Fig. 4. Specification of MMU⋆'s page swap

5.3 The MMU⋆ case study

Our third benchmark is MMU⋆ that uses IFC policies to isolate application memory resources. Our implementation is inspired by [Choudhuri and Givargis 2005], that introduces the principle of a software Memory Management Unit (MMU) for embedded devices that lack hardware virtualization.

MMU⋆ allows us to use virtualization to isolate tasks but also to permit data transfer between tasks memory. When writing to a memory location, a program must craft a virtual address with a `tag`: the target task's id, a `page_index`: the page to write, and an `offset_index`. This allows any tasks to read and write on any memory location depending on the policy.

In order to perform the transaction to physical memory, the MMU⋆ translates the virtual address to a physical one using a private function called `translation`. This function takes two arguments: a virtual address and a boolean that indicates the type of translation (read or write). Figure 4, illustrate the translation algorithm.

Our example, has two tasks: `task1` which writes two bytes in the memory of the second task and `task2` which reads these two bytes and performs an addition. We use LIO⋆ to show that `task2` reads only its own memory, in a way similar to BUS⋆ presented before (§ 5.2).

In this example, as in BUS⋆, we observe C code reduction and runtime speedup when IFC checks are statically proved. As Table 5 presents our implementation is 124 LoP in all versions, while LoC greatly reduced from 192 LoC in DLIO⋆, to 176 LoC in SLIO⋆, and to

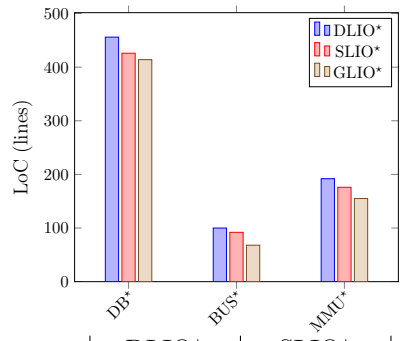
155 LoC in GLIO⋆. As with BUS⋆, the C code reduction leads to runtime speedups of 23% for SLIO⋆ and 47.7% for GLIO⋆ shown in Table 7. Thus, again we get important runtime speedup with low verification effort.

5.4 Evaluation

Finally we summarize the implementation platform and evaluation of our libraries.

Figure 6 presents the hardware and software for the development and evaluation of our libraries. We used the GNU C Compiler (GCC) compiler because our work is targeting micro-controller like Amtel which are not supported by the verified compiler CompCert. For our benchmarks, we explicitly disable any compiler optimizations.

Figure 5 summarizes the LoF and the extracted Lines of C Code (LoC) for each of our three benchmarks. Figure 7 summarizes the running times for each of the benchmark and the speedup of each library version with respect to DLIO⋆. To measure these times, we run the BUS⋆ and the MMU⋆ benchmarks $75M$ times and the DB⋆ benchmarks $20k$ times. Concerning data set, DB⋆ has a database with 2000 papers where 1992 belong to the user Mary, the BUS⋆ uses a list of packet that test all combination of the lattice labels, and the MMU⋆ doesn't need a dataset.



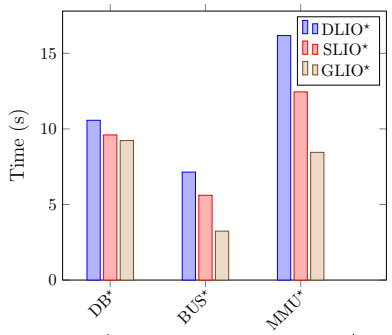
	DLIO⋆		SLIO⋆		GLIO⋆	
	LoF	LoC	LoF	LoC	LoF	LoC
DB⋆ (§ 5.1)	166	456	265	426	269	414
BUS⋆ (§ 5.2)	94	100	94	92	94	68
MMU⋆ (§ 5.3)	124	192	124	176	124	155
Total	384	748	483	694	487	637

Fig. 5. Size measurement of all presented uses cases.

GCC	8.2.1 20181215	Processor	Intel(R) Xeon(R) W-2104
Fedora	5.3.6-100.fc29.x86_64	Frequency	3.2 GHz
F⋆	aeb4203c841735a6f401ed8b9cd44412a68c82bb	Core	4
KreMLin	882534387830a4cc259c1a543e0dfc10dcc70f52	Architecture	x86_64
		RAM	16GB

Fig. 6. Details our benchmark platform

We conclude that moving from DLIO⋆, to SLIO⋆, and to GLIO⋆, is expected to increase the LoF but reduces the LoC, since IFC is statically checked. Since the lines of C code are reduced and runtime checks and label book keeping is removed, this leads to runtime speedups. From our three benchmarks we conclude that this extra effort in static verification pays off in low-level applications, like MMU⋆ and BUS⋆ that use few but critical IFC checks. On the other hand, in applications like DB⋆ that are heavy on dynamic IFC checks the verification effort is high and the effect on runtime improvement small.



	DLIO⋆	SLIO⋆		GLIO⋆	
	Time	Time	Speed-up	Time	Speed-up
DB⋆ (§ 5.1)	10.571s	9.602s	9.1%	9.229s	12.6%
BUS⋆ (§ 5.2)	7.144s	5.610s	21.4%	3.238s	54.6%
MMU⋆ (§ 5.3)	16.179s	12.449s	23%	8.450s	47.7%

Fig. 7. Performance measurement of all presented uses cases.

6 RELATED WORKS

LIO⋆ descends from a vast line of works started from the basis of MAC in [Bell and LaPadula 1973] to the general lattice-theoretic model proposed by Denning in [Denning and Denning 1977] to verify the control information flow on computer or operating systems.

Today, large, security-sensitive, applications use expensive and state-of-the-art architectures, such as Risc-V or ARM designs, to implement systematic resource isolation at supervisor and hardware-MMU kernel level to safely sandbox legacy services of large corporate or cloud infrastructures. The open-source RISC-V architecture supports extensions providing hardware IFC capabilities encoded as a byte-size tag alongside with data [Ferraiuolo et al. 2018; Palmiero et al. 2018] to control data flow in accordance with the tags privileges. ARM's Trustzone allows to segregate encrypted and decrypted data in physically isolated trust zones. [De Amorim et al. 2015] generalizes this meta-data tag mechanism to implement more general software-defined IFC policies at hardware level.

Virtualization technology and resource isolation available in modern operating systems and verified micro-kernels [Gu et al. 2016; Klein et al. 2009] is however far from available to consumer-market, IoT-oriented, embedded micro-controller architectures. On such targets, compartmentalization is a cost-effective compilation technique to complement label-enforced IFC policy with defensive code to isolate possible software faults and prevent program threads from addressing data outside of their designated partitions [Besson et al. 2019; De Amorim et al. 2015].

Software-defined IFC helps to overcome hardware limitations and can, where available, strengthen coarse-grain, hardware security mechanisms (trust zones, virtualization, tags) with fine-grain user-, task- or channel-level micro-policies [De Amorim et al. 2015]. Software-level IFC was first proposed in [Myers 1999b] to annotate Java programs with IFC policies. [Hammer et al. 2006] provides a language-agnostic library to check IFC properties in imperative C or Java programs.

Dynamic IFC policies have extensively been developed in operating system design. [Zeldovich et al. 2006] provides a survey covering this domain. For instance, [Krohn et al. 2007] proposes operating system mechanisms to systematically check information flow read or

written by system threads. LIO is introduced in [Giffin et al. 2012] and showed in [Parker et al. 2019] to support dynamic IFC for web applications using a verified implementation in Liquid Haskell. In [Buiras et al. 2015a], LIO mixes static and dynamic verification in Haskell to mitigate the constraints of runtime checks.

IFC [Sabelfeld and Myers 2006] policies can be used to ensure component separation, but current techniques that enforce such policies either use heavy runtime checks [Austin and Flanagan 2012; Austin et al. 2017; Yang et al. 2016] or rely on advanced type-checking using high level programming languages [Buiras et al. 2015a; Schoepe et al. 2014].

LIO relies on Haskell's monads to effectively enforce IFC policies when reading/writing to databases or the web [Parker et al. 2019; Stefan et al. 2017]. [Gregersen et al. 2019] presents the implementation of a statically verified IFC policy in Idris. Idris is a pure functional programming language with dependent type and proof assistance. [Gregersen et al. 2019] shows how modelling IFC using dependent types improves expressibility of IFC policies.

Hence, direct application of software-defined IFC policies to embedded devices with, *e.g.,* LIO, faces two major obstacles. First, LIO's policy enforcement relies on automatically generated runtime checks that would, if not properly sand-boxed, cause an unattended device to crash unpredictably. Second, garbage collection and lazy evaluation in high-level languages may overflow their limited memory if implemented without extensive engineering, testing and profiling efforts.

Our approach takes advantage of both the expressivity of dependent-types in the verified programming language F* [Swamy et al. 2016], allowing us to use F*'s effects to encode monadic IFC encapsulation, and the capability of generating bare-metal system code, by using its KreMLin [Protzenko et al. 2017] code generator. This approach yields three advantages:

Like related approaches based on high-level programming languages, [Buiras et al. 2015a; Gregersen et al. 2019; Parker et al. 2019; Stefan et al. 2017], LIO* lifts policy enforcement from runtime checks to static proof obligations by using powerful dependent-type systems using a minimalistic effect system defined from F*'s Dijkstra Monads [Maillard et al. 2019], leading to verified code with minimal mechanisation, and in similar ways to [Parker et al. 2019], using Liquid Haskell's refinement type system.

[Buiras et al. 2015a] offers a different hybridation mechanism that ours: it eliminates IFC runtime checks that can be ruled safe statically and keep other, call-dependent, dynamic checks. This is a most suitable approach for transactional applications, where throwing an exception from some LIO client application is non-critical or fail-safe. However, in the case of, possibly unattended, reactive applications, this is not an option, as failing safe usually means to restart a real-time and potentially mission- or safety-critical application.

[Sabelfeld and Russo 2010] provides a detailed review on the extensive number of related approaches based on the static analysis of imperative system programs. The recent [Guarnieri et al. 2020], for instance, statically analyses bytecode to monitor programs that may leak unintended information when executed on speculative architectures. As in these approaches, LIO* offers the capability to run verified code generated from the KreMLin compiler [Protzenko et al. 2017], without the need for a runtime library or a garbage collector, and hence for direct application for low-level, resource-constrained, embedded architectures.

[Gregersen et al. 2019] models LIO in the domain-specific language Idris to mechanically verify its specification by theorem proving; and [Parker et al. 2019] uses the refinement reflection mechanism of Liquid Haskell [Vazou et al. 2017] to automate the proof for a model of LIO in the λ-calculus. LIO* verifies the actual LIO*'s client applications correct and non-interfering by using F*'s meta-programming environment Meta* [Martínez et al. 2019].

[Sabelfeld and Russo 2010] shows static verification of IFC policies to be as strong as its, more permissive, dynamic enforcement via defensive code or monitors, both static and dynamic approaches ultimately providing the same assurance of termination-insensitive non-interference.

7 CONCLUSION AND FUTURE WORKS

We presented LIO⋆ a verified F⋆, IFC framework. First, we implemented three library versions: 1) the dynamic DLIO⋆, where IFC policies are checked at runtime, 2) the static SLIO⋆, where IFC policies are lifted to compile time proof obligations, and 3) the ghost GLIO⋆, where the IFC label tracking is totally erased at runtime. Next, we used metaprogramming to define `genNILemma`, a procedure that generates a lemma stating that a concrete GLIO⋆ client is noninterferent. We applied `genNILemma` to two of our benchmarks and various smaller examples, and generated noninterference lemmata that F⋆ automatically proved correct. We evaluated our approach using three benchmarks and validated that moving from DLIO⋆, to SLIO⋆, to GLIO⋆, the verification effort (*i.e.,* the F⋆ Lines of Code) increases, but the extracted C code is cleaner (and shorter) and up to %54.6 faster (in the case of BUS⋆). In general, we propose a methodology that is using static verification to reduce runtime checks and metaprogramming to prove program metaproperties, leading to both fast and provably correct software, ideal to be executed by embedded devices.

Future works

Metaproperties. To verify our methodology, we developed a mechanized noninterference proof of GLIO⋆'s clients. As discussed in § 4.4 our approach suffers from various limitations, some of them stem from Meta⋆, which prevented us from proving noninterference of the DB⋆ clients. As Meta⋆ gets more mature and our experience with it grows, we aim to complete the metatheory of DB⋆ in the near future and even apply this metaprogramming for metaproperties methodology to generalize a noninterference proof for every GLIO⋆ client.

Layered Effects. Our current implementation of LIO⋆ suffers from a lack of parametricity: it is not possible to use our framework without, for instance, Low⋆'s `ST` effect, the standard F⋆ way of writing low-level programs. Layered effect, a new F⋆ feature, addresses this issue.

This last improvement of F⋆ will enable us to define IFC effects independently of target IO computation (memory, streams, etc) and allow to express IFC policies in a much simpler and elegantly modular way by the composition of effects. This will make our IFC library a lot more portable and allow to significantly reduce its TCB.

Our current implementation of LIO⋆ only handles pure functions and we couldn't yet use Low⋆ effects to extend it with a complete memory model, which would have been much of an improvement for, *e.g.,* the MMU⋆ case study. There are several reasons why we made the design choice to limit our working prototype with pure effects: 1/ using F⋆'s base effects would have implied modifying the Low⋆ API to include labels, 2/ furthermore, the Low⋆ memory model and API are going to be revamped (C code extraction from KreMLin doesn't currently play nicely with reifiable effect), and 3/ the notion of layered effect, a new way of compositionally declaring effects, is currently being tested in F⋆.

This last improvement of F⋆ will enable us to define IFC effects independently of target IO monad (memory, streams, etc) and allow to express IFC policies in a much simpler and elegantly modular way by the composition of effects. This will make our IFC library a lot more portable and allow us to significantly reduce its TCB.

REFERENCES

Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martinez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. 2017. Dijkstra Monads for Free. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 515–529. https://doi.org/10.1145/3009837.3009878

Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/2103656.2103677

Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst.* 39, 3, Article 10 (May 2017), 56 pages. https://doi.org/10.1145/3024086

D Elliott Bell and Leonard J LaPadula. 1973. *Secure computer systems: Mathematical foundations*. Technical Report. MITRE CORP BEDFORD MA.

Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. 2019. Compiling Sandboxes: Formally Verified Software Fault Isolation. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 499–524.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015a. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 289–301.

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015b. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/2784731.2784758

Siddharth Choudhuri and Tony Givargis. 2005. Software virtual memory management for MMU-less embedded systems. *Center for Embedded Computer Systems* (2005), 12.

Arthur Azevedo De Amorim, Maxime Dénes, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 813–830.

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243. https://doi.org/10.1145/360051.360056

Dorothy E Denning and Peter J Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (1977), 504–513.

Andrew Ferraiuolo, Mark Zhao, Andrew C Myers, and G Edward Suh. 2018. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1583–1600.

Andrzej Filinski. 1994. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 446–457. https://doi.org/10.1145/174675.178047

Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. 2012. Hails: Protecting data privacy in untrusted web applications. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 47–60.

Simon Gregersen, Søren Eller Thomsen, and Aslan Askarov. 2019. A dependently typed library for static information-flow control in idris. In *International Conference on Principles of Security and Trust*. Springer, 51–75.

Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent {OS} Kernels. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 653–669.

Marco Guarnieri, Boris Köpf, Jose Morales, Jan Reineke, and Andrés Sánchez. 2020. SPECTECTOR: Principled Detection of Speculative Information Flows. In *Security and Privacy Conference*. IEEE.

Christian Hammer, Jens Krinke, and Gregor Snelting. 2006. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*. IEEE Computer Press, 87–96.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 207–220.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *ACM SIGOPS Operating Systems*

Review, Vol. 41. ACM, 321–334.

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cundefinedtundefinedlin Hriundefinedcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proc. ACM Program. Lang.* 3, ICFP, Article 104 (July 2019), 29 pages. https://doi.org/10.1145/3341708

Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP)*. Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

Andrew C. Myers. 1999a. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Andrew C Myers. 1999b. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 228–241.

Christian Palmiero, Giuseppe Di Guglielmo, Luciano Lavagno, and Luca P Carloni. 2018. Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 1–7.

James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information Flow Security for Multi-tier Web Applications. *Proc. ACM Program. Lang.* 3, POPL, Article 75 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290388

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. https://doi.org/10.1145/3110261

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-Weight Information-Flow Security in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1411286.1411289

A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Andrei Sabelfeld and Alejandro Russo. 2010. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In *Perspectives of Systems Informatics*, Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 352–365.

Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. 2014. SeLINQ: Tracking Information across Application-Database Boundaries. *SIGPLAN Not.* 49, 9 (Aug. 2014), 25–38. https://doi.org/10.1145/2692915.2628151

Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. 2017. Flexible Dynamic Information Flow Control in the Presence of Exceptions. *Journal of Functional Programming* 27 (2017).

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Eran Tromer and Roei Schuster. 2016. DroidDisintegrator: Intra-Application Information Flow Control in Android Apps. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 401–412. https://doi.org/10.1145/2897845.2897888

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *ACM Trans. Comput. Logic* 4, 1 (Jan. 2003), 1–32. https://doi.org/10.1145/601775.601776

Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *Proceedings of*

the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 631–647. https://doi.org/10.1145/2908080.2908098

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 263–278.