Building secure systems with LIO (demo)

Citation for the original published paper (version of record):
Stefan, D., Levy, A., Russo, A. et al (2014). Building secure systems with LIO (demo). ACM SIGPLAN Notices, 49(12): 93-94. http://dx.doi.org/10.1145/2633357.2633371

Building Secure Systems with LIO (Demo)

Deian Stefan[1] Amit Levy[1] Alejandro Russo[2] David Mazières[1]

[1] Stanford University [2] Chalmers University of Technology

{deian, alevy, ⊥}@cs.stanford.edu russo@chalmers.se

Abstract

LIO is a decentralized information flow control (DIFC) system, implemented in Haskell. In this demo, we give an overview of the LIO library and show how LIO can be used to build secure systems. In particular, we show how to specify high-level security policies in the context of web applications, and describe how LIO automatically enforces these policies even in the presence of untrusted code.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Security; LIO; DCLabels; Hails; Decentralized information flow control; Web application

1. Introduction

Haskell provides many language features that can be used to reduce the damage caused by any particular piece of code. Notable among these are the strong static type system and module system. The type system, in addition to reducing undefined behavior, can be used to distinguish between pure and side-effecting computations, i.e., computations that respectively can and cannot affect the "external world," while the module system can be used to enforce abstraction (e.g., by restricting access to constructors).[1] Unfortunately, even in such a high-level, type-safe language, building software systems is an error-prone task and only a few programmers are equipped to write secure code.

Consider, for instance, a conference review system where reviewers are expected to be anonymous and users in conflict with a paper are prohibited from reading specific committee comments. When building such a system, if we import a library function that performs IO, we risk violating these guarantees—if the code is malicious, it may, for instance, read reviews from the database and leak them to a public server. Worse yet, such code may be leaking information through more subtle means, e.g., by encoding data in the number of reviews. How, then, can we restrict the effects of a computation, without imposing that it not perform *any* side-effects?

One approach is to restrict computations to a particular monad—one other than `IO`—for which we can control effects. In this demonstration, we describe the LIO library which implements one such monad, called `LIO` (*Labeled IO*) [6, 7]. Effects in the `LIO` monad are mediated according to decentralized information flow control (DIFC) policies [3, 4]. In particular, this means that computations can perform arbitrary effects, as long as they do not violate the confidentiality or integrity of data. (Indeed, LIO automatically disallows effects that would violate confidentiality or integrity.)

[1] Here, we refer to the safe subset of the Haskell language—without `unsafePerformIO`, etc.—as enforced by the *Safe Haskell* extension [9].

Haskell '14, September 4–5 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2633357.2633371

2. Overview

DIFC systems such as LIO track and control the propagation of information by associating a *label* with every piece of data. (While LIO is polymorphic in the label model, we focus on LIO with DCLabels [5], henceforth just labels.) A label encodes a security policy as a pair of positive boolean formulas over *principals* specifying who may read or write data. For example, a review labeled `"alice" \/ "bob" %% "bob"` specifies that the review can be read by user `"alice"` or `"bob"`, but may only be modified by `"bob"`. Indeed, such a label may be associated with `"bob"`'s review, for a paper that both `"bob"` and `"alice"` are reviewing.

Our LIO library associates labels with various Haskell constructs. For example, we provide labeled alternatives of `IORef`, `MVar`, and `Chan`, called `LIORef`, `LMVar`, and `LChan`, respectively. Moreover, we provide an implementation of a filesystem that associates persistent labels with files and a type, `Labeled`, that is used to associate a label with individual Haskell terms. The latter, for example, is used to associate labels with reviews (e.g., as given by the type `Labeled DCLabel Review`).

Labels on objects are partially ordered according to a *can flow to* relation $\sqsubseteq$: for any labels L_A and L_B, if $L_A \sqsubseteq L_B$ then the policy encoded by L_A is *upheld* by that of L_B. For example, data labeled L_A = `"alice" \/ "bob" %% "bob"` can be written to a file labeled L_B = `"bob" %% "bob"` since L_B preserves the secrecy of L_A. In fact, L_B is *more* restrictive, as only `"bob"`—not both `"alice"` and `"bob"`—can read the file, and, indeed, until `"alice"` submits her review we may wish to associate this label with `"bob"`'s review as to ensure that she cannot read it. Conversely, $L_B \not\sqsubseteq L_A$, and thus data labeled L_B cannot be written to an object labeled L_A (data secret to `"bob"` cannot be leaked to a file that `"alice"` can also read).

It is precisely this relation that is used by LIO when restricting the effects performed by a computation in the `LIO` monad. In fact, the `LIO` monad solely encapsulates the underlying `IO` computation and a label, called the *current label*, that tracks the sensitivity of the data that the computation has observed. To illustrate the role of the current label, consider the code below that reads `"bob"`'s private review and tries to leak it into a reference that `"alice"` can read.

```
-- Current label: public == True %% True
bobReview <- readFile "/reviews/bob/5.txt"
-- Current label: "bob" %% True
-- labelOf aliceRef == "alice" %% "alice"
writeLIORef aliceRef $ show bobReview
-- Fail: "bob" %% True ⋢ "alice" %% "alice"
```

Here, the current label is first raised by `readFile` to reflect the fact that information sensitive to `"bob"` was incorporated into the context. Importantly, however, this label is also used to subsequently restrict the effects performed by the computation; in this case, the `writeLIORef` action raises an exception to reflect that the computation tried to write to a reference whose label does not protect the review content.

In general, DIFC enforcement in LIO follows this approach of exposing functions (e.g., `writeLIORef`), which inspect the current label and the label of object they are about to read/write as to uphold the *can flow to* relation. Our definition for bind and return are trivial; we solely rely on Haskell's monad support as a way to define a sublanguage that enforces DIFC. By ensuring (with Safe Haskell) that untrusted code is written in this sublanguage, i.e., it cannot lift arbitrary `IO` actions into `LIO`, we can incorporate arbitrary (untrusted) code to compute on sensitive data. For example, our conference review system can incorporate code provided by users of the system without fear of leaking reviews or reviewer identities, all while allowing the code to interact with the external world.

A further important consequence of this approach to DIFC is that once we have a sound core language, which, in the case of LIO, is both concurrent and supports exceptions, [2] we can introduce many features by simply wrapping existing `IO` code. As mentioned, LIO supports labeled alternatives to mutable references, mutable variables, channels, files, databases, HTTP clients, etc. Some of these features (e.g., the database) are crucial for implementation applications such as the conference review system.

3. Automatic data labeling for Web applications

LIO guarantees that code executing in the `LIO` monad cannot violate the confidentiality and integrity restrictions imposed by labels. Unfortunately, assigning appropriate labels to data is challenging and setting overly-permissive labels can amount to unexpected "leaks." While using a simple label model such as DCLabels may help avoid certain pitfalls, an alternative approach is clearly desirable.

In the context of web applications, we present an advancement towards making DIFC policy-specification a mortal task.[3] Specifically, we demonstrate the declarative policy language, previously developed for the Hails web framework [1]. In web applications, it is common for developers to specify the application data model in a declarative fashion. Hails leverages this design parading and the observation that, in many web applications, the authoritative source for who should access data resides in the data itself to provide developers with a means for specifying the policy alongside the data model.

Consider the definition of the `Review` data type used in our conference review system:

```
data Review = Review { reviewId    :: ReviewId
                     , reviewPaper :: PaperId
                     , reviewOwner :: UserName
                     , reviewBody  :: Text }
```

To associate a label with a review we can leverage the information present in the record type. Specifically, we can specify that the only user allowed to modify such a review is the owner of the review herself; and, we can specify that the only users allowed to read such a review are the owner and other reviewers of the same paper. The latter declaration requires that we perform a lookup, using the paper id of the current review, to find the other reviewers. The code implementing this policy is given below.

```
policy :: HailsDB m => Review -> m DCLabel
policy rev = do
  let author = reviewOwner rev
  reviewers <- findReviewersOf $ reviewPaper rev
  makePolicy $ do
    readers ==> author \/ reviewers
    writers ==> author
```

The function is self-explanatory; we only remark that the function takes a `Review` and returns a `DCLabel` in a monad `m` that allows code to perform database actions (in this case the `findReviewersOf` action), a change from the original pure policies of Hails.

We remark, that while, some care must be taken to ensure that the specified policy is correct, the extend to understanding a security policy in such LIO/Hails applications is limited to such functions. It is these policy functions that the database system uses to label reviews when a fetch, insert, or update is performed. Indeed, the core of the conference review system does not manipulate labels—high-level APIs make most of the DIFC details transparent.

4. Demonstration

The demonstration will explain the basics of DIFC and how LIO can be used to enforce information flow security on untrusted code. In particular, we will show how the core of a simple, web-based conference review system is implemented in LIO. Part of this includes the specification of high-level policies, which is facilitated by the use of the simple DCLabels model and our automatic labelling paradigm. To demonstrate the flexibility of our automatic labeling we will show how arbitrary untrusted code can be used to replace the core busy-logic of the application.

Acknowledgements This work was funded by DARPA CRASH under contract #N66001-10-2-4088. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

[1] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *Proc. of the 10th OSDI*, pages 47–60. USENIX, 2012.

[2] C. Hriţcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your ifcexception are belong to us. In *Proc. of the IEEE Symp. on Security and Privacy*, 2013.

[3] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th SOSP*, pages 129–142, 1997.

[4] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[5] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec 2011*, LNCS. Springer, 2011.

[6] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, 2011.

[7] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the 17th ICFP*, 2012.

[8] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[9] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe haskell. In *ACM SIGPLAN Notices*, volume 47, pages 137–148. ACM, 2012.

[2] The presence of exceptions in the core calculus is very important, since it allows code to recover from DIFC violation attempts [2, 8]. For example, the failure of the above code to write to a reference is not fatal—the untrusted code can recover and continue executing.

[3] We considered the alternative approach, cloning MIT Prof. N. Zeldovich.