Dynamic Enforcement of Dynamic Policies

Pablo Buiras and Bart van Delft

Chalmers University of Technology, Sweden

Abstract. LIO is a dynamic information-flow control system embedded in Haskell that uses a runtime monitor to enforce noninterference. The monitor is written as a library, requiring no changes to the runtime. We propose to extend LIO with a state component, allowing us to enforce not only noninterference but also information-flow policies that change while the program is running.

Enforcement mechanisms for information flows in software frequently aim to achieve the *noninterference* security property. This property states that no change in sensitive (secret) inputs to the system should affect non-sensitive (public) outputs, which captures the idea that secrets should not be leaked.

LIO [3] is a Haskell runtime monitor that enforces noninterference. Over time, LIO has been successfully extended to prevent information leaks via certain covert timing channels. The security condition has, however, not yet been generalised, even though it is generally accepted that noninterference is too strong a requirement for most applications.

There are several canonical examples of applications that necessarily violate noninterference. A password checker needs to allow for some interference from the password database to the user to signal whether the login attempt was successful or not. Information purchase applications require noninterference on confidential information to hold only until the price for that information has been paid. Yet other applications might need to introduce additional noninterference constraints over time, for example on the information flow from strategic documents to a manager who is demoted while the system is running.

To allow for the enforcement of such dynamic policies, we propose to extend LIO with a state component which records that part of the system state relevant to determine the current policy that needs to be enforced. In the following we briefly summarise how LIO works and how we propose to extend it. For now we consider only the original sequential LIO library, leaving support for extensions such as concurrency to future work.

Labelled IO LIO leverages Haskell's monadic encoding of side-effects to provide security. In Haskell, input/output operations are provided by the `IO` *monad*, an abstract data type used to express sequencing of effectful computations. The `LIO` monad provided by the LIO library is intended to be used as a replacement for this type. It provides a collection of operations similar to `IO`, but enriched with security checks that prevent unwanted information flows. `LIO` computations carry the type `LIO l a`, where `l` is an arbitrary security lattice of labels specified by the code using LIO and `a` is the type of the result of the computation.

The LIO library uses a *floating-label* approach to the dynamic enforcement of information-flow policies, which is based on mandatory access control. The `LIO` monad uses its state to keep track of a *current label*, $L_{\texttt{cur}}$. This label represents, in a coarse-grained way, the least upper bound over the labels on which the current computation depends. All the (I/O) operations provided by `LIO` take care to appropriately validate and adjust this label. Consider a standard two-point lattice (`Low` $\sqsubseteq$ `High`) and a computation starting with $L_{\texttt{cur}}$ being `Low`. When this computation reads a file labelled `High`, $L_{\texttt{cur}}$ is raised to `High` and writing to `Low` files is prohibited by the `LIO` monad from that moment onwards, independent of what would actually be written to these files.

Stateful LIO We propose for LIO computations to carry the type `LIO s l a`, where `s` is the type of the state component for `LIO` to use in its enforcement. That is, when writing to a file with label l we now check whether $L_{\texttt{cur}} \sqsubseteq_s l$. The `LIO` library exports functionality to update this state s, so the outcome of this check for the same $L_{\texttt{cur}}$ and l can vary depending on the current value of s.

As the relation between labels can now change arbitrarily over time, the labels lose their lattice structure and a least upper bound can no longer be computed. Therefore $L_{\texttt{cur}}$ is modified to contain the *set* of labels of all the information on which the current computation depends. When performing a sensitive operation like writing to a file, the $\sqsubseteq_s$ check is performed for each label in $L_{\texttt{cur}}$ individually.

Encodings We can now present various policy change mechanisms as restricted interfaces to Stateful LIO. Clearly, we can regain the original noninterference by simply not exporting the operations to update the state component.

We can export an explicit `declassify` function, by using a boolean value as the state component and having the ordering among policies as usual except that `High` $\sqsubseteq_{true}$ `Low` holds but `High` $\sqsubseteq_{false}$ `Low` does not. An operation `p` can now be declassified by calling `declassify p` which sets the state to *true*, performs `p` and then resets the state to *false* before returning.

We can also encode policy languages that allow for much more policy change, such as Paralocks [1] (in which the state component becomes a set of open locks) or non-disclosure policies [2] (where the state tracks the set of flow-relations).

References

1. Niklas Broberg and David Sands. Paralocks – Role-Based Information Flow Control and Beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.
2. Ana Almeida Matos and Gerard Boudol. On declassification and the non-disclosure policy. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 226–240. IEEE, 2005.
3. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.