# A Library for Secure Multi-threaded Information Flow in Haskell

Ta-chung Tsai Alejandro Russo John Hughes
Department of Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden

## Abstract

*Li and Zdancewic have recently proposed an approach to provide information-flow security via a library rather than producing a new language from the scratch. They show how to implement such a library in Haskell by using arrow combinators. However, their approach only works with computations that have no side-effects. In fact, they leave as an open question how their library, and the mechanisms in it, need to be modified to consider side-effects. Another absent feature in the library is support for multithreaded programs. Information-flow in multithreaded programs still remains an open challenge, and no support for that has been implemented yet. In this light, it is not surprising that the two main stream compilers that provide information-flow security, Jif and FlowCaml, lack support for multithreading.*

*Following ideas taken from literature, this paper presents an extension to Li and Zdancewic's library that provides information-flow security in presence of reference manipulation and multithreaded programs. Moreover, an online-shopping case study has been implemented to evaluate the proposed techniques. The case study reveals that exploiting concurrency to leak secrets is feasible and dangerous in practice. To the best of our knowledge, this is the first implemented tool to guarantee information-flow security in concurrent programs and the first implementation of a case study that involves concurrency and information-flow policies.*

## 1 Introduction

Language-based information flow security aims to guarantee that programs do not leak confidential data, by some form of static analysis which rejects programs that would leak, before they are run. Over the years, a great many such systems have been presented, supporting a wide variety of programming constructs [26]. However, the impact on programming practice has been rather limited.

One possible reason is that most systems are presented in the context of a simple, elegant, and minimal language, with a well-defined semantics to make proofs of soundness possible. Yet such systems cannot immediately be adopted by programmers—they must first be embedded in a real programming language with a real compiler, which is a major task in its own right. Only two such languages have been developed—Jif [14, 15] (based on Java) and FlowCaml [19, 30] (based on Caml).

Yet when a system implementor chooses a programming language, information flow security is only one factor among many. While Jif or FlowCaml might offer the desired security guarantees, they may be unsuitable for other reasons, and thus not adopted. This motivated Li and Zdancewic to propose an alternative approach, whereby information flow security is provided via a *library* in an existing programming language [13]. Constructing such a library is a much simpler task than designing and implementing a new programming language, and moreover leaves system implementors free to choose any language for which such a library exists.

Li and Zdancewic showed how to construct such a library for the functional programming language Haskell. The library provides an abstract type of secure programs, which are composed from underlying Haskell functions using operators that impose information-flow constraints. Secure programs are *certified*, by checking that all constraints are satisfied, before the underlying functions are invoked—thus guaranteeing that no secret information leaks. While secure programs are a little more awkward to write than ordinary Haskell functions, Li and Zdancewic argue that typically only a small part of a system need manipulate secret data—for example, an authentication module—and only this part need be programmed using their library.

However, Li and Zdancewic's library does impose quite severe restrictions on what a secure program fragment may do. In particular, these fragments may have no effects of any sort, since the library only tracks information flow through the inputs and outputs of each fragment. While absence of side-effects can be guaranteed in Haskell (via the type system), this is still a strong restriction. Our purpose in this