# Secure Multi-Execution in Haskell

Mauro Jaskelioff and Alejandro Russo

[1] CIFASIS-CONICET/Universidad Nacional de Rosario.
[2] Dept. of Computer Science and Engineering, Chalmers University of Technology

**Abstract.** Language-based information-flow security has emerged as a promising technology to guarantee confidentiality in on-line systems, where enforcement mechanisms are typically presented as run-time monitors, code transformations, or type-systems. Recently, an alternative technique, called *secure multi-execution*, has been proposed. The main idea behind this novel approach consists on running a program multiple times, once for each security level, using special rules for I/O operations. Compared to run-time monitors and type-systems, secure multi-execution does not require to inspect the full code of the application (only its I/O actions). In this paper, we propose the core of a library to provide *non-interference* through secure-multi execution. We present the code of the library as well as a running example for Haskell. To the best of our knowledge, this paper is the first work to consider secure-multi execution in a functional setting and provide this technology as a library.

## 1 Introduction

Over the past years, there has been a significant increase in the number of online activities. Users can do almost everything using a web browser. Even though web applications are probably among the most used pieces of software, they suffer from vulnerabilities that permit attackers to steal confidential data, break the integrity of systems, and affect the availability of services. Web-based vulnerabilities have already outplaced those of all other platforms [1] and there are no reasons to think that this situation is going to change [9].

In this work, we focus on preserving confidentiality of data through the security policy known as non-interference [3, 10] (i.e. not leaking secrets into public channels). Confidentiality policies are getting more and more relevant for widely open connected systems as the web, where compromised confidential data can be used to impersonate users in Facebook, Twitter, Flickr, and other social networks.

Language-based information-flow security [27] has developed approaches to analyze applications' code, leading to special-purpose languages, interpreters or compilers [19, 24] that guarantee security policies like non-interference. Rather than producing new languages from scratch, security can also be provided by libraries [14]. The potential of this approach has been shown across a range of programming languages and security policies [32, 25, 23, 4, 15, 6].

Traditionally, information-flow analysis on a program is done statically (e.g. using a type-system), dynamically (e.g. using an execution monitor), or with a combination of both. Recently, authors in [7] devised an alternative approach, called *secure multi-execution*, based on the idea of executing the same program several times, once for

each security level. As opposed to previous enforcement mechanisms, this novel approach does not demand to design type-systems or deploy heavy-weight monitoring of programs; it only requires modifying the semantics of I/O operations.

In this paper, we present the main ideas of a library based on monads [17, 16] to provide *non-interference* through secure-multi execution. The ideas can be easily applied to any pure language and are illustrated with an implementation for the programming language Haskell. To the best of our knowledge, this paper is the first one to consider secure-multi execution as library in a pure functional setting.

## 2 Secure multi-execution

Devriese and Piessens [7] propose the novel approach of secure multi-execution to enforce non-interference. We organize security levels in a security lattice $\mathcal{L}$, where security levels are ordered by a partial order $\sqsubseteq$, with the intention to only allow leaks from data at level $\ell_1$ to data at level $\ell_2$ when $\ell_1 \sqsubseteq \ell_2$. Secure multi-execution runs a program multiple times, once for each security level. In order to enforce security, the I/O operations of those multiple copies of the program are interpreted differently. Outputs on a given channel at security level $\ell$ is performed only in the execution of the program linked to that security level. Inputs coming from a channel at security level $\ell$ are replaced by a default value if the execution of the program is linked to a security level $\ell_e$ such that $\ell \not\sqsubseteq \ell_e$. In that manner, the execution of the program linked to level $\ell_e$ never obtains information higher than its security level. In the case that $\ell_e = \ell$, the input operation is performed normally. Finally, if $\ell \sqsubseteq \ell_e$, the execution of the program linked to level $\ell_e$ reuses the inputs obtained by the execution linked to level $\ell$.

Devriese and Piessens show that secure multi-execution is *sound* and *precise*. Soundness states that each execution linked to a given level cannot get any information from higher levels and consequently, all of its output will have to be generated from information at its level or below, guaranteeing non-interference. Precision establishes that if a program satisfies non-interference under normal execution, then its behavior is the same as the one obtained by secure multi-execution on terminating runs.

## 3 Secure multi-execution in Haskell

In most pure functional programming languages, computations with side-effects such as inputs and outputs can be distinguished by its type. For instance, in Haskell every computation performing side-effects must be encoded as a value of the monad (or abstract data type) $IO$ [22]. Specifically, a value of type $IO\ \ a$ is an action (i.e., a computation which may have side-effects) which produces a value of type $a$ when executed. This manner in which monads identify computations with side-effects fits particularly well with the idea of secure multi-execution of giving different interpretations to I/O operations as specified by the execution level.

For simplicity, we consider a two-point security lattice with elements $L$ and $H$, where $L \sqsubseteq H$ and $H \not\sqsubseteq L$. Levels $L$ and $H$ represent public and secret confidentiality levels, respectively. The implementation shown here, however, works for an arbitrary

finite security lattice. In Fig. 1 we show the implementation of the lattice as elements of the datatype $Level$ and define the order relationship $\sqsubseteq$ and the non-reflexive $\sqsubset$.

$$\begin{aligned}
&\mathbf{data}\ Level = L \mid H\ \mathbf{deriving}\ (Eq, Enum)\\
&\cdot \sqsubseteq \cdot, \cdot \sqsubset \cdot :: Level \to Level \to Bool\\
&H \sqsubseteq L = False\\
&_ \sqsubseteq\ _ = True\\
&p \sqsubset q\ = p \sqsubseteq q \wedge p \not\equiv q
\end{aligned}$$

**Fig. 1.** Security lattice

We propose a library that works by replacing I/O actions (i.e., values of the $IO$ monad) by a pure description of them [31]. Haskell programs which perform some I/O actions have type $a \to IO\ b$. That is, given some argument of type $a$, the program performs some I/O actions and then returns a value of type $b$ as the result. In secure multi-execution, the I/O actions performed by such program must be interpreted differently depending on the security level linked to a given execution (see Section 2). Hence, programs to be run under secure multi-execution do not return I/O actions, but rather a pure description of them. With this in mind, secure programs have the type $a \to ME\ b$, where monad $ME$ describes the side-effects produced during the computation. When the program is executed, those I/O descriptions are interpreted according to the specification of secure multi-execution. For security levels $L$ and $H$, the program is run twice, where the I/O actions are interpreted differently on the execution linked at level $L$, and on the one linked at level $H$. Figure 2 summarizes the ideas behind our library. Function $run$ executes and links the program to the security level given as argument. Observe that function $run$ is also responsible for the interpretation of the I/O actions described in the monad $ME$.

For simplicity, we only consider reading and writing files as the possible I/O actions. It is easy to generalize our approach to consider other I/O operations. Function $level :: FilePath \to Level$ assigns security levels to files indicating the confidentiality of their contents. We assume that, when a file is read, its access time gets updated as a side-effect of the operation. An attacker, or public observer, is able to learn the content of public files as well as their access time.

a) $a \longrightarrow IO\ b$

b) $a \longrightarrow ME\ b \xrightarrow[run\ H]{run\ L} IO\ b$

c) $\mathbf{data}\ ME\ a = Return\ a$
$\quad \mid Write\ FilePath\ String\ (ME\ a)$
$\quad \mid Read\ FilePath\ (String \to ME\ a)$

**Fig. 2.** Type for a typical program with side-effects (a) and a secure multi-execution program (b), and definition of $ME$ (c).

Monad $ME$ describes the I/O actions performed by programs and is defined in Fig. 2(c). Constructors $Return$, $Write$, and $Read$ model programs performing different actions. Program $Return\ x$ simply returns value $x$ without performing any I/O operations. Program $Write\ file\ x\ p$ models a program that writes string $x$ into file *file* and then behaves as program $p$. Program $Read\ file\ g$ models a program that reads the contents $x$ of file *file* and then behaves as program $g\ x$. Technically, $ME$ is an intermediate monad that provides a pure model of the reading and writing of files in the $IO$ monad.

Users of the library do not write programs using the constructors of $ME$ directly. Instead, they use the interface provided by the monad: $return :: a \to ME\ a$ and

$(\gg\!=) :: ME\ a \to (a \to ME\ b) \to ME\ b$. The $return$ function lifts a pure value into the $ME$ monad. The operator $\gg\!=$, called *bind*, is used to sequence computations. A bind expression $(m \gg\!= f)$ takes a computation $m$ and function $f$ which will be applied to the *value* produced by $m$ and yields the resulting computation. These are the only primitive operations for monad $ME$, and consequently, programmers must sequence individual computations explicitly using the bind operator. Fig. 3 shows the implementation of $return$ and $\gg\!=$. The expression $return\ x$ builds a trivial computation, i.e., a computation which does not perform any $Write$/$Read$ actions and just returns $x$. Values in $ME$ are introduced with $Return$, so this is the only case where $f$ is applied. In the other two cases, the $Write$/$Read$ operations are preserved and bind with $f$ $(\gg\!= f)$ is recursively applied. Besides $return$ and $(\gg\!=)$, the monad $ME$ has operations to denote I/O actions on files. These operations model the equivalent operations on the $IO$ monad and are given by the following functions.

$$
\begin{array}{l}
\textbf{instance}\ Monad\ ME\ \textbf{where} \\
\quad return\ x \qquad\qquad\ = Return\ x \\
\quad (Return\ x) \quad \gg\!= f \quad = f\ x \\
\quad (Write\ file\ s\ p) \gg\!= f = Write\ file\ s\ (p \gg\!= f) \\
\quad (Read\ file\ g) \gg\!= f \quad = Read\ file\ (\lambda i \to g\ i \gg\!= f)
\end{array}
$$

**Fig. 3.** Definitions for $return$ and $\gg\!=$

$$
\begin{array}{l}
writeFile :: FilePath \to String \to ME\ () \\
writeFile\ file\ s = Write\ file\ s\ (return\ ())
\end{array}
\qquad
\begin{array}{l}
readFile :: FilePath \to ME\ String \\
readFile\ file = Read\ file\ return
\end{array}
$$

## 4 An interpreter for the monad *ME*

$$
\begin{array}{l}
run :: Level \to ChanMatrix \to ME\ a \to IO\ a \\
run\ l\ \_\ (Return\ a) = return\ a \\
run\ l\ c\ (Write\ file\ o\ t) \\
\quad \mid level\ file \equiv l \quad = \textbf{do}\ IO.writeFile\ file\ o \\
\qquad\qquad\qquad\qquad\quad\ run\ l\ c\ t \\
\quad \mid otherwise \qquad = run\ l\ c\ t \\
run\ l\ c\ (Read\ file\ f) \\
\quad \mid level\ file \equiv l \quad = \textbf{do}\ x \leftarrow IO.readFile\ file \\
\qquad\qquad\qquad\qquad\quad\ broadcast\ c\ l\ file\ x \\
\qquad\qquad\qquad\qquad\quad\ run\ l\ c\ (f\ x) \\
\quad \mid level\ file \sqsubset l \quad = \textbf{do}\ x \leftarrow reuseInput\ c\ l\ file \\
\qquad\qquad\qquad\qquad\quad\ run\ l\ c\ (f\ x) \\
\quad \mid otherwise \qquad = run\ l\ c\ (f\ (defvalue\ file))
\end{array}
$$

**Fig. 4.** Interpreter for monad $ME$

Fig. 4 shows the interpreter for programs of type $ME\ a$. Intuitively, $run\ l\ c\ p$ executes $p$ and links the execution to security level $l$. Argument $c$ is used when inputs from executions linked to lower levels need to be reused (explained below). The implementation is pleasantly close to the informal description of secure multi-execution in Section 2. Outputs are only performed ($IO.writeFile\ file\ o$) when the confidentiality level of the output file is the same as the security level linked to the execution ($level\ file \equiv l$). Inputs are obtained ($IO.readFile\ file$) when files' confidentiality level is the same as the security level linked to the execution ($level\ file \equiv l$). Data from those inputs is broadcasted to executions linked to higher security levels in order to be properly reused when needed ($broadcast\ c\ l\ file\ x$). If the current execution level is higher than the file's confiden-

tiality level ($level\ file \sqsubset l$)), the content of the file is obtained from the execution linked to the same security level as the file ($reuseInput\ \ c\ \ l\ \ file$). Otherwise, the input data is replaced by a default value. Function $defvalue :: FilePath \rightarrow String$ sets default values for different files. Unlike [7], and to avoid introducing runtime errors, we adopt a default value for each file (i.e., input point) in the program. Observe that inputs can be used differently inside programs. For instance, the contents of some files could be parsed as numbers, while others as plain strings. Therefore, choosing a constant default value, e.g. the empty string, could trigger runtime errors when trying to parse a number out of it.

An execution linked to security level $\ell$ reuses inputs obtained in executions linked to lower levels. Hence, we implement communication channels between executions, from a security level $\ell'$ to a security level $\ell$, if $\ell' \sqsubset \ell$. In the interpreter, the argument of type $ChanMatrix$ consists of a matrix of communication channels indexed by security levels. An element $c_{\ell',\ell}$ of the matrix denotes a communication channel from security level $\ell'$ to $\ell$ where $\ell' \sqsubset \ell$; otherwise $c_{\ell',\ell}$ is undefined. In this manner, execution linked at level $\ell'$ can send its inputs to the execution linked at level $\ell$, where $\ell' \sqsubset \ell$. Messages transmitted on these channels have type $(FilePath, String)$, i.e., pairs of a filename and its contents. Function $broadcast\ \ c\ \ l\ \ file\ \ x$ broadcasts the pair $(file, x)$ on the channels linked to executions at higher security levels, i.e., channels $c_{l,\ell}$ such that $l \sqsubset \ell$. Function $reuseInput\ \ c\ \ l\ \ file$ matches the filename $file$ as the first component of the pairs in channel $c_{level\ file,l}$ and returns the second component, i.e., the contents of the file.

```
sme :: ME a → IO ()
sme t = do
  c ← newChanMatrix
  l ← newEmptyMVar
  h ← newEmptyMVar
  forkIO (do run L c t; putMVar l ())
  forkIO (do run H c t; putMVar h ())
  takeMVar l; takeMVar h
```

**Fig. 5.** Secure multi-execution

Multithreaded secure multi-execution is orchestrated by the function $sme$. This function is responsible for creating communication channels to implement the reuse of inputs, creating synchronization variables to wait for the different threads to finish, and, for each security level, forking a new thread that runs the interpreter at that level. Fig. 5 shows a specialized version of $sme$ for the two-point security lattice. However, in the library implementation, the function $sme$ works for an arbitrary finite lattice. Function $newChanMatrix$ creates the communication channels. Synchronization variables are just simple empty $MVar$s. When a thread tries to read ($takeMVar$) from an empty $MVar$ it will block until another thread writes to it ($putMVar$) [21]. Function $forkIO$ spawns threads that respectively execute the interpreter $run$ at levels $L$ and $H$, and then signal termination by writing ($putMVar\ \ l\ \ (); putMVar\ \ h\ \ ()$) to the thread's synchronization variable. The main thread locks on these variables by trying to read from them ($takeMVar\ \ l; takeMVar\ \ h$).

Unlike [7], function $sme$ does not require the scheduler to keep the execution at level $L$ ahead of the execution at level $H$. In [7], this requirement helps to avoid timing leaks at the price of probably modifying the runtime system (i.e, the scheduler). As mainstream information-flow compilers, monad $ME$ also ignores timing leaks.

```
data CreditTerms = CT { discount :: Rational, ddays :: Rational, net :: Rational }
calculator :: ME ()
calculator = do loanStr  ← readFile "Client"
                termsStr ← readFile "Client-Terms"
                let loan     = read loanStr
                    terms    = read termsStr
                    interest = loan − loan ∗ (1 − discount terms / 100)
                    disct    = discount terms / (100 − discount terms)
                    ccost    = disct ∗ 360 / (net terms − ddays terms)
                writeFile "Client-Interest" (show interest)
                writeFile "Client-Statistics" (show ccost)
                  -- writeFile "Client-Statistics" (show ccost ++ loanStr)
```

**Fig. 6.** Financial calculator

## 5 A motivating example

We present a small example of how to build programs using monad $ME$. We consider the scenario of a financial company who wants to preserve the confidentiality of their clients but, at the same time, compute statistics by hiring a third-party consultant company. Given certain loan, the company wants to write code to compute the *cost of credit* [5] and the total amount of interest that it will receive as income. When taking a loan, credit terms usually indicate a due date as well as a cash discount if the credit is canceled before an expiration date. We consider credit terms of the form "*discount* / *discount period* net / *credit period*", which indicates that if payment is made within *discount period* days, a *discount* percent cash discount is allowed. Otherwise, the entire amount is due in *credit period* days. Given a credit term, the amount of money paid when the credit is due is $loan - loan \times (1 - discount/100)$. The yearly cost of credit, i.e., the cost of borrowing money under certain terms is $\frac{discount}{100-discount} \times \frac{360}{credit\ period - discount\ period}$. For instance, in an invoice of \$1000 with terms 2 /10 net 30, the total interest payed at the due date is $\$1000 - \$1000 \times (1 - .2) =$ \$20, and the cost of credit becomes $\frac{2}{98} \times \frac{360}{20} = .3673$, i.e., $37\%$.

In this setting, we consider the amount of every loan to be confidential (secret) information, while cost of credit is public and thus available for statistics. By writing our program using monad $ME$, we can be certain that confidential information is never given for statistics. In other words, the third-party consultant company does not learn anything about the amount in the loans provided by the financial company. Figure 6 shows one possible implementation of the program to compute interests and cost of credit. Files `"Client"` and `"Client-Interest"` are considered secret (level $H$), while `"Client-Terms"` and `"Client-Statistics"` are considered public (level $L$). The code is self-explanatory.

If a programmer writes, by mistake or malice, $show\ ccost +\!\!+ loanStr$ as the information to be written into the public file (see commented line), then secure multi-execution avoids leaking the sensitive information in $loanStr$ by given the empty string to the execution linked to security level $L$.

## 6 Related work

Previous work addresses non-interference and functional languages [11, 33, 24, 29]. The seminal work by Li and Zdancewic [14] shows that information-flow security can also be provided as a library for real programming languages. Morgenstern et al. [18] encode a programming language aware of authorization and information-flow policies in Agda. Devriese and Piessens [8] enforce non-interference, either dynamically or statically, using monad transformers in Haskell. Different from that work, the monad $ME$ does not encode static checks in the Haskell's type-system or monitor every step of programs' executions. Moreover, Devriese and Piessens' work requires to encode programs as values of a certain data type, while our approach only models I/O operations.

Russo et al. [26] outline the ground idea for secure multi-execution as a naive transformation. A transformed program runs twice: one execution computes the public results, where secret inputs were removed, and the second execution computes the secret outputs of the program. Devriese and Piessens [7] propose secure multi-execution as a novel approach to enforce non-interference. Devriese and Piessens implement secure multi-execution for the Spider-monkey JavaScript engine. The implementation presented in this work is clean and short (approximately 130 lines of code), and thus making it easy to understand how multi-execution works concretely. Unlike [7], our approach does not consider termination and timing covert channels. We argue that dealing with termination and timing covert channels in a complex language, without being too conservative, is a difficult task. In this light, it is not surprising that the main information-flow compilers (Jif [20] –based on Java–, and FlowCaml [30] –based on Caml–) ignore those channels.

Close to the notion of secure multi-execution, Jif/split [34] automatically partitions a program to run securely on heterogenously trusted hosts. Different from secure multi-execution, the partition of the code is done to guarantee that if host $h$ is subverted, hosts trusting $h$ are the only ones being compromised. Swift [2] uses Jif/split technology to partition the program into JavaScript code running on the browser, and JavaScript code running on the web server.

## 7 Concluding remarks

We propose a monad and an interpreter for secure multi-execution. To the best of our knowledge, we are the first ones to describe secure multi-execution in a functional language. We implement our core ideas in a small Haskell library of around 130 lines of code and present a running example. The implementation is compact and clear, which makes it easy to understand how secure multi-execution works concretely. Broadcasting input values to executions at higher levels is a novelty of our implementation if compared with Devriese and Piessens' work. This design decision is not tied to the Haskell implementation, and the idea can be used to implement the reuse of inputs in any secure multi-execution approach for any given language. The library is publicly available [13].

*Future work* Our long-term goal is to provide a fully-fledged library for secure multi-execution in Haskell. The $IO$ monad can perform a wide range of input and output

operations. It is then interesting to design a mechanism capable to lift, as automatically as possible, $IO$ operations into the monad $ME$ [12]. Another direction for future work is related with declassification, or deliberate release of confidential information [28]. Declassification in secure multi-execution is still an open challenge. Due to the structure of monadic programs, we believe that it is possible to identify, and restrict, possible synchronization points where declassification might occur. Then, declassification cannot happen arbitrarily inside programs but only on those places where we can give some guarantees about the security of programs. To evaluate the capabilities of our library, we plan to use it to implement a medium-size web application. Web applications are good candidates for case studies due to their demand on confidentiality as well as frequent input and output operations (i.e. server requests and responses). It is also our intention to perform benchmarks to determine the overhead introduced by our library. The library seems to multiply execution time by the number of levels, but since file operations are only done once, the reality could be better if the broadcast mechanism is not expensive.

*Acknowledgments* Alejandro Russo was partially funded by the Swedish research agency VR. We would like to thank Arnar Birgisson, Andrei Sabelfeld, Dante Zanarini and the anonymous reviewers for their helpful comments.

## References

[1] M. Andrews. Guest Editor's Introduction: The State of Web Security. *IEEE Security and Privacy*, 4(4):14–15, 2006.

[2] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, Oct. 2007.

[3] E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.

[4] J. J. Conti and A. Russo. A taint mode for Python via a library. *NordSec 2010. Selected paper by OWASP AppSec Research 2010*, 2010.

[5] Credit Research Foundation. Ratios and formulas in customer financial analysis. `http://www.crfonline.org/orc/cro/cro-16.html`, 1999.

[6] F. Del Tedesco, A. Russo, and D. Sands. Implementing erasure policies using taint analysis. In T. Aura, editor, *The 15th Nordic Conf. in Secure IT Systems*. Springer Verlag, 2010.

[7] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[8] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '11, pages 59–72, New York, NY, USA, 2011. ACM.

[9] Federal Aviation Administration (US). Review of Web Applications Security and Intrusion Detection in Air Traffic Control Systems. `http://www.oig.dot.gov/sites/dot/files/pdfdocs/ATC_Web_Report.pdf`, June 2009. Note: thousands of vulnerabilities were discovered.

[10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[11] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 365–377, Jan. 1998.

[12] M. Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

[13] M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. software release. `http://www.cse.chalmers.se/~russo/sme/`, 2011.

[14] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[15] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In T. Aura, editor, *The 15th Nordic Conf. in Secure IT Systems*. Springer Verlag, 2010.

[16] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Edinburgh, Scotland, 1989.

[17] E. Moggi. Computational lambda-calculus and monads. In *Proc., Fourth Annual Symposium on Logic in Computer Science*, pages 14–23. IEEE Computer Society, 1989.

[18] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN Int. Conf. on Funct. Prog.*, ICFP '10, pages 169–180, New York, NY, USA, 2010. ACM.

[19] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, Jan. 1999.

[20] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[21] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *POPL '96: Proc. of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, New York, NY, USA, 1996. ACM.

[22] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. of the ACM Conf. on Principles of Programming*, pages 71–84, 1993.

[23] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In R. Safavi-Naini and V. Varadharajan, editors, *ACM Symposium on Information, Computer and Communications Security (ASIACCS 2009)*, Sydney, Australia, March 2009. ACM Press.

[24] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, Jan. 2002.

[25] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell'08: Proc. of the 1st ACM SIGPLAN Symp. on Haskell*. ACM, 2008.

[26] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Asian Comp. Science Conf. (ASIAN'06)*, LNCS. Springer-Verlag, 2007.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[28] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[29] V. Simonet. Flow caml in a nutshell. In *Graham Hutton, editor, Proc. of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[30] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.

[31] W. Swierstra and T. Altenkirch. Beauty in the beast. In *Proc. of the ACM SIGPLAN workshop on Haskell*, Haskell '07, pages 25–36, New York, NY, USA, 2007. ACM.

[32] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, July 2007.

[33] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, July 2002.

[34] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proc. ACM Symp. on Operating System Principles*, 2001.