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 [Security 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.

```haskell
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 functionality in another project. Bob’s code has the following type signature.

```haskell
Bob
commonpwds :: 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.

```haskell
Alice
import qualified Bob as Bob
password :: IO String
password = do
  putStr "Please, select your password:
  pwd ← getLine
  b ← Bob.commonpwds pwd
  if b then putStrLn "It’s a common 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.

```haskell
Bob
commonpwds 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 legal action for function common\_pwd\_s (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\_pwd\_s 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.txt", and "dict\_en\_txt.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\_pwd\_s, 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 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 $\ell_1$ can flow into entities labeled with $\ell_2$. 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., $L \sqsubseteq H$ and $H \not\sqsubseteq L$. Figure 1 shows the encoding of this two-point lattice using type classes (Russo et al. 2008) ①. 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 compromised②. The index for this family consists on a security label $\ell$ 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 $MAC^{\_TCB}$

① Orphan instances could break the security lattice. Readers should refer to the accompanying source code to learn how to avoid that.

② 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.
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 $\text{TCB}$. Function $\text{io}^{\text{TCB}}$ lifts arbitrary $\ell$-actions into the security monad. The definitions for return and bind are straightforward. Function $\text{run}^{\text{MAC}}$ 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 $\text{MAC} \ H \ (\text{IO} \ \text{String})$ is probably an insecure computation—the IO-action could be arbitrary and reveal secrets, e.g., consider the code $\text{return } "\text{secret}" \Rightarrow \lambda h \rightarrow \text{return} \ \langle \text{http://bob.evil/pwd=} \ + \ h \rangle$.

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. The safe 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 $\ell$ (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) $\ell$. 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 $\ell_1$ and $\ell_2$ to assert that $\ell_1 \subseteq \ell_2$.

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 $d$ as opposed to $a$ will become evident when extending the library with secure versions of existing data types (e.g., Section 3).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3}
\caption{Labeled resources}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4}
\caption{Interaction between MAC $\ell$ and labeled resources.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5}
\caption{Synthesizing secure functions by mapping read and write effects to security checks}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6}
\caption{Labeled expressions}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7}
\caption{Secure interaction between family members}
\end{figure}

$\text{data} \ Id \ a = \text{Id}^{\text{TCB}} \ \{ \text{unId}^{\text{TCB}} :: a \}$

$\text{type} \ Labeled \ a = \text{Res} \ (\text{Id} \ a)$

$\text{label} :: \ell_1 \subseteq \ell_2 \Rightarrow a \rightarrow \text{MAC} \ \ell_1 \ \text{(Labeled } \ell_2 \ a)$

$\text{label} = \text{new}^{\text{TCB}} \ \text{return} \ . \ \text{Id}^{\text{TCB}}$

$\text{unlabel} :: \ell_1 \subseteq \ell_2 \Rightarrow \text{Labeled} \ \ell_1 \ a \rightarrow \text{MAC} \ \ell_2 \ a$

$\text{unlabel} = \text{read}^{\text{TCB}} \ \text{return} \ . \ \text{unId}^{\text{TCB}}$

$\text{read}^{\text{TCB}} :: \ell_1 \subseteq \ell_2 \Rightarrow$

$(d \ a \rightarrow \text{IO} \ a) \rightarrow \text{MAC} \ \ell_1 \ (d \ a) \rightarrow \text{MAC} \ \ell_2 \ a$

$\text{read}^{\text{TCB}} \ f \ (\text{Res}^{\text{MAC}} \ da) = (\text{io}^{\text{TCB}} . f) \ da$

$\text{write}^{\text{TCB}} :: \ell_1 \subseteq \ell_2 \Rightarrow$

$(d \ a \rightarrow \text{IO} () \rightarrow \text{Res} \ \ell_1 \ (d \ a) \rightarrow \text{MAC} \ \ell_1 ()$

$\text{write}^{\text{TCB}} \ f \ (\text{Res}^{\text{MAC}} \ da) = (\text{io}^{\text{TCB}} . f) \ da$

$\text{new}^{\text{TCB}} :: \ell_1 \subseteq \ell_2 \Rightarrow \text{IO} \ (d \ a) \rightarrow \text{MAC} \ \ell_1 \ (\text{Res} \ \ell_2 \ (d \ a))$

$\text{new}^{\text{TCB}} \ f = \text{io}^{\text{TCB}} \ f \Rightarrow \text{return} \ . \ \text{Res}^{\text{TCB}}$

\footnote{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.}
actions can suppress subsequent read effects from monad \( a \), e.g., it is easy for \( MAC \) to read from a resource labeled as \( a \) such that \( a \subseteq a \). Nevertheless, data obtained from such reads has no evident effect for monad \( MAC \). Observe that, by type-checking, sensitive data acquired in \( MAC \) cannot be used to build actions in \( MAC \). In other words, from the perspective of \( String\)⇒\( a \) readIORef \( String\) to read from a resource labeled as \( a \)-actions when calling \( pwds \)-actions \( H \) does not violate the no read-up and no write-down rules for monad \( MAC \). Operationally, function \( join \) which satisfies the no write-down rule for monad \( MAC \) has no evident effect for monad \( MAC \). Even \( H \) runs the computation of type \( MAC \) such that \( H \subseteq a \), it is like those read effects have never occurred. With respect to write effects, monad \( MAC \) is allowed to write into labeled resources at sensitivity \( a \) such that \( a \subseteq a \). By the type constrain in \( join \) and transitivity, it holds that \( a \subseteq a \), which satisfies the no write-down rule for monad \( MAC \).

Despite trusting our types to reason about \( join \), there exists a subtlety that escapes the power of Haskell’s type-system and can compromise security: the integration of non-terminating \( MAC \)-actions can suppress subsequent \( MAC \)-actions. By detecting that certain actions never occurred, \( MAC \) can infer that non-terminating \( MAC \)-actions are triggered by \( join \). If such non-terminating actions were triggering depending on secret values, \( MAC \) could learn about sensitive information. Sections 4 and 6 describe how to adapt the implementation of \( join \) to account for this problem—for now, readers should assume terminating \( MAC \)-actions when calling \( join \).

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 \( MAC \). 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.

![Figure 8. Secure references](image)

code has access to \( MAC \)’s internals and removes the constructor \( Res \) wrapping the boolean. Alice now has guarantees that Bob’s code is not leaking secrets.

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 \( common\_pwd\) 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 10 monad which implements memoization.

![Table 1](image)

The code marks the password as sensitive \( (pwd) \), runs Bob’s code, and obtains the result \( (\text{bias}) \)—since Alice is trustworthy, her

---

3 e.g., by applying appropriate lifting operations (Swamy et al. 2011)
resources. We introduce the type \( \text{Ref}^\text{MAC} \ell \ a \) as a type synonym for \( \text{Res} \ell \ (\text{IOR} \ a) \)—see Figure 8. Secondly, we consider functions \( \text{newIOR} \ell \ a \rightarrow \text{IO} \ (\text{IOR} \ a) \), \( \text{readIOR} \ell \ a \rightarrow \text{IO} \ a \), \( \text{writeIOR} \ell \ a \rightarrow \text{IO} \ell \) 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 \( \text{newIORef} \ell \ a \rightarrow \text{IO} \ (\text{IOR} \ a) \), \( \text{readIORef} \ell \ a \rightarrow \text{IO} \ a \), \( \text{writeIORef} \ell \ a \rightarrow \text{IO} \ell \) are lifted into the monad \( \text{MAC} \ell \) by wrapping them using \( \text{new}^\text{MC} \), \( \text{read}^\text{MC} \), and \( \text{write}^\text{MC} \), respectively. We remark that these steps naturally generalize to obtain secure interfaces of various kinds. (For instance, Section 6 shows how to add \( \text{MVars} \) by applying similar steps.) With secure references available in \( \text{MAC} \), 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 \( \text{MAC} \ell \).

\[
\begin{align*}
\text{mem}^\text{MAC} : & : (\text{String} \rightarrow \text{MAC} \ell \ \text{String}) \\
& \rightarrow \text{MAC} \ell \ (\text{String} \rightarrow \text{MAC} \ell \ \text{String})
\end{align*}
\]

We leave the implementation of this function as an exercise for the reader\(^8\). Bob also generalizes \( \text{common} \cdot \text{pwd} \) to be parametric in the function used to fetch URLs.

\[
\begin{align*}
\text{common} \cdot \text{pwd} : & : (\text{String} \rightarrow \text{MAC} \ell \ \text{String}) \rightarrow \text{wget} \\
& \rightarrow \text{Labeled} \ H \ \text{String} \\
& \rightarrow \text{MAC} \ell \ (\text{Labeled} \ H \ \text{Bool})
\end{align*}
\]

Finally, Alice puts all the pieces together by initializing the memoized version of \( \text{wget}^\text{MAC} \) and pass it to \( \text{common} \cdot \text{pwd} \).

\[
\begin{align*}
\text{Alice} & : \begin{array}{l}
\text{password} : \ : \ (\text{IO} \ \text{String}) \\
\text{password} & = \ : \ (\text{actable}) \ (\text{mem}^\text{MAC} \ \text{wget}^\text{MAC}) \\
\text{askWith} & = \ : \ (\text{wget}^\text{Mem}) \\
\text{askStr} "\text{Please, select your password:"} \\
\text{pwd} & = \rightarrow \text{getLime} \\
\ell \text{pwd} & = \rightarrow \text{label} \ \text{pwd} : \ : \ (\text{MAC} \ell \ (\text{Labeled} \ H \ \text{String})) \\
\ell \text{bool} & = \rightarrow \text{run}^\text{MC} \ (\ell \text{pwd} \rightarrow \ell \text{bool}) \\
\ell \text{b} & = \ : \ (\text{unRes} \ \ell \text{bool}) \\
\ell \text{bool} & = \rightarrow \text{label} \ \ell \text{bool} \\
\text{else} & \rightarrow \text{return} \ \ell \text{pwd}
\end{array}
\end{align*}
\]

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 \( \text{MAC} \) 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 uncaugh exception thrown by \( \text{wget}^\text{MAC} \). Clearly, \( \text{MAC} \) 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 \( \text{Maybe} \). Pure computations are all that programmers need in this case—a feature already supported by \( \text{MAC} \). More interestingly, Haskell allows throwing exceptions anywhere, but only catching them within the \( \text{IO} \) monad. To extend \( \text{MAC} \) with such a system, we need to lift exceptions and their operations to securely work in monad \( \text{MAC} \ell \).

Figure 9 shows functions \( \text{throw}^\text{MAC} \) and \( \text{catch}^\text{MAC} \) to throw and catch secure exceptions, respectively. Exceptions can be thrown anywhere within the monad \( \text{MAC} \ell \). 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 \( \text{join}^\text{MAC} \) and exceptions is quite subtle. As the next example shows, their interaction might lead to compromised security.

**Example 5.** Alice extends \( \text{MAC} \) with the primitives in Figure 9. Tired of dealing with Bob, she asks Charlie to help him adapt his code to recover from failures in \( \text{wget}^\text{MAC} \). 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.

\[
\begin{align*}
\text{Alice} & : \begin{array}{l}
\text{password} : \ : \ (\text{IO} \ \text{String}) \\
\text{password} & = \ : \ (\text{actable}) \ (\text{mem}^\text{MAC} \ \text{wget}^\text{MAC}) \\
\text{askWith} & = \ : \ (\text{wget}^\text{Mem}) \\
\text{askStr} "\text{Please, select your password:"} \\
\text{pwd} & = \rightarrow \text{getLime} \\
\ell \text{pwd} & = \rightarrow \text{label} \ \text{pwd} : \ : \ (\text{MAC} \ell \ (\text{Labeled} \ H \ \text{String})) \\
\ell \text{bool} & = \rightarrow \text{run}^\text{MC} \ (\ell \text{pwd} \rightarrow \ell \text{bool}) \\
\ell \text{b} & = \ : \ (\text{unRes} \ \ell \text{bool}) \\
\ell \text{bool} & = \rightarrow \text{label} \ \ell \text{bool} \\
\text{else} & \rightarrow \text{return} \ \ell \text{pwd}
\end{array}
\end{align*}
\]

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 \( \text{MAC} \) 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 uncaugh exception thrown by \( \text{wget}^\text{MAC} \). Clearly, \( \text{MAC} \) needs support to recover from such errors.

Function \( \text{Ex4} \cdot \text{common} \cdot \text{pwd} \) 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 \( \text{join}^\text{MAC} \) block could stop the production of a subsequent public event.

\[
\begin{align*}
\text{Alice} & : \begin{array}{l}
\text{password} : \ : \ (\text{IO} \ \text{String}) \\
\text{password} & = \ : \ (\text{actable}) \ (\text{mem}^\text{MAC} \ \text{wget}^\text{MAC}) \\
\text{askWith} & = \ : \ (\text{wget}^\text{Mem}) \\
\text{askStr} "\text{Please, select your password:"} \\
\text{pwd} & = \rightarrow \text{getLime} \\
\ell \text{pwd} & = \rightarrow \text{label} \ \text{pwd} : \ : \ (\text{MAC} \ell \ (\text{Labeled} \ H \ \text{String})) \\
\ell \text{bool} & = \rightarrow \text{run}^\text{MC} \ (\ell \text{pwd} \rightarrow \ell \text{bool}) \\
\ell \text{b} & = \ : \ (\text{unRes} \ \ell \text{bool}) \\
\ell \text{bool} & = \rightarrow \text{label} \ \ell \text{bool} \\
\text{else} & \rightarrow \text{return} \ \ell \text{pwd}
\end{array}
\end{align*}
\]

Defined as \( \text{join}^\text{MAC} \), function \( \text{proxy} : : \ell \rightarrow \text{MAC} \ell \) is used to fix the family member involved in the code enclosed by \( \text{join}^\text{MAC} \). The code crashes if the secret boolean is true (\( \text{bool} \equiv \text{True} \)); otherwise, it sends a http-request to Bob’s server indicating that the secret is false (\( \text{http://bob.evil/bit=f} \)).

By using \( \text{catch}^\text{MAC} \), Bob implements malicious code capable of leaking one bit of sensitive data.
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.

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

The reason for the attack is the use of `MAC` \( \ell_{l} \)-actions which can suppress subsequent `MAC` \( \ell_{l} \)-actions by simply throwing exceptions (see `join^{MAC}` in function `crashOnTrue`). As the attack shows, exceptions can be thrown at the least sensitive family members (Stefan et al. 2012b). Unfortunately, types are of little help here: on one hand, `join^{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^{MAC}` to rule out Bob’s attack.

We redefine `join^{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^{MAC}`. It runs the computation \( m \) while catching any possible raised exception. Importantly, `join^{MAC}` returns a value of type `Labeled \( \ell_{l} \)` even if exceptions are present. In case of abnormal termination, `join^{MAC}` returns a labeled value which contains an exception—this exception is re-thrown when forcing its evaluation. In the definition of `join^{MAC}`, function `slabel` is used instead of `label` in order to avoid introducing type constraint `slabel :: Labeled Bool`.

```
join^{MAC} :: \ell_{l} \subseteq \ell_{h} \Rightarrow
      MAC \ell_{h} a \rightarrow MAC \ell_{l} (Labeled \ell_{h} a)
```

```
where

\begin{align*}
\text{slabel} &= \text{return} . \text{Res}^{TCB} . \text{Id}^{TCB}\end{align*}
```

**Figure 10.** Revised version of `join^{MAC}`

\( \ell_{h} \subseteq \ell_{l} \). Interested readers can verify that if \( \ell_{h} \subseteq \ell_{l} \) is a tautology (as it is the case in `MAC`), the implementation of `slabel` and `label` are equivalent in `join^{MAC}`.

**Example 6.** Before Bob could deploy his attack, Alice submits the revised version of `join^{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^{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.

```
attack :: Labeled H String \rightarrow MAC L ()
attack lpwd =
  do
    toChars lpwd \equiv mapM charToByte \Rightarrow
      return ()
  common lweds uget lpwd >>
    attack lpwd \equiv Ex4 common lweds uget lpwd
```

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 con-
tents and lengths—see definition for dict. Then, for each generated password \( \text{forM dict (guess lpwd)} \), function guess asserts if it is equal to the password under scrutiny \( \text{pwd} \equiv \text{try} \). 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.

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 IPC 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 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
forkMAC :: MAC ℓ () → MAC ℓ ()
forkMAC = ioTCP . forkIO . runMAC
```

Function \( \text{forkMAC} \) 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 \( \text{joinMAC} \) and \( \text{forkMAC} \) 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 function structurally similar to \( \text{crashOnTrue} \), i.e., containing a \( \text{joinMAC} \) block followed by a public event.

```
Boob
loopOn :: Bool → L → L ()
loopOn True lbool n = do
  loopOn False lbool n
  return ()
```

Function \( \text{loopOn} \) loops if the secret coincides with its first argument. Otherwise, it sends the value \( \neg \text{try} \) to Bob’s server. As the next step, Bob takes the attack from Section 4 and modifies function \( \text{leakBit} \) as follows.

```
Bob
leakBit :: Lbool → L → L (),
leakBit lbool n =
  forkMAC (loopOn True lbool n) \_implies
  forkMAC (loopOn False lbool n) \_implies
  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 \( \ell \_\text{true} \)-action can suppress the execution of subsequently MAC \( \ell \_\text{false} \)-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 \( \text{joinMAC} \) (see function \( \text{loopOn} \)) suppresses the execution of \( \text{wgetMAC} \) 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 \( \text{joinMAC} \) by \( \text{forkMAC} \) 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., \( MAC \ell \_\text{true} \), because the decision to do so depends on data at level \( \ell \_\text{true} \). Although we remove \( \text{joinMAC} \), 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 \( \text{forkMAC} \) with the one in Figure 11 and removes \( \text{joinMAC} \) from the API. As an immediate result of that, function \( \text{loopOn} \) does not compile any longer. The only manner for \( \text{loopOn} \) to inspect the secret and perform a public side-effect is by replacing \( \text{joinMAC} \) with \( \text{forkMAC} \) as follows.

\[
\text{forkMAC} :: \ell \_\text{true} \implies \ell \_\text{false} \implies \ell \_\text{false} \implies \ell \_\text{false} \implies \text{return} ()
\]
Synchronization Primitives

Synchronization primitives are vital for concurrent programs. In 6.1 Synchronization Primitives, we consider MACs as labeled resources, where type synonym MVar MAC ℓ a is defined as Res ℓ ( MVar a), see Figure 12. Secondly, we obtain secure version of functions newEmptyMVar MAC :: ℓ1 ⊆ ℓ4 ⇒ MAC ℓ1 ( MVar MAC ℓ2 a), newEmptyMVar MAC = newTCB newEmptyMVar MAC.

takeMVar MAC :: (ℓ1 ⊆ ℓ3, ℓ2 ⊆ ℓ4) ⇒ MVar MAC ℓ1 a → MAC ℓ1 a

takeMVar MAC = wrT CB takeMVar

putMVar MAC :: (ℓ4 ⊆ ℓ5, ℓ1 ⊆ ℓ4) ⇒ MVar MAC ℓ1 a → a → MAC ℓ1 ()

putMVar MAC lmv v = wrT CB (flip putMVar v) lmv

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 MVars—an established synchronization abstraction in Haskell (Peyton Jones et al. 1996).

We proceed in a similar manner as we did for references. We consider MVars as labeled resources, where type synonym MVar MAC ℓ a is defined as Res ℓ ( MVar a), see Figure 12. Secondly, we obtain secure version of functions newEmptyMVar MAC :: ℓ1 ⊆ ℓ4 ⇒ MAC ℓ1 ( MVar MAC ℓ2 a), newEmptyMVar MAC = newTCB newEmptyMVar MAC.

takeMVar MAC :: (ℓ1 ⊆ ℓ3, ℓ2 ⊆ ℓ4) ⇒ MVar MAC ℓ1 a → MAC ℓ1 a

takeMVar MAC = wrT CB takeMVar

putMVar MAC :: (ℓ4 ⊆ ℓ5, ℓ1 ⊆ ℓ4) ⇒ MVar MAC ℓ1 a → a → MAC ℓ1 ()

putMVar MAC lmv v = wrT CB (flip putMVar v) lmv

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.

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


---

\(^8\) Number obtained with the software measurement tool SLOCCount


