Securing Asynchronous Exceptions

Carlos Tomé Cortiñas
Chalmers University of Technology
carlos.tome@chalmers.se

Marco Vassena
CISPA Helmholtz Center for Information Security
marco.vassena@cispa.saarland

Alejandro Russo
Chalmers University of Technology
russo@chalmers.se

***Abstract*—Language-based information-flow control (IFC) techniques often rely on special purpose, ad-hoc primitives to address different covert channels that originate in the runtime system, beyond the scope of language constructs. Since these piecemeal solutions may not compose securely, there is a need for a unified mechanism to control covert channels. As a *first step* towards this goal, we argue for the design of a general interface that allows programs to safely interact with the runtime system and the available computing resources. To coordinate the communication between programs and the runtime system, we propose the use of asynchronous exceptions (*interrupts*), which, to the best of our knowledge, have not been considered before in the context of IFC languages. Since asynchronous exceptions can be raised at any point during execution—often due to the occurrence of an external event—threads must temporarily *mask* them out when manipulating locks and shared data structures to avoid deadlocks and, therefore, breaking program invariants. Crucially, the naive combination of asynchronous exceptions with existing features of IFC languages (e.g., concurrency and synchronization variables) may open up new possibilities of information leakage. In this paper, we present MAC$_{async}$, a concurrent, statically enforced IFC language that, as a novelty, features asynchronous exceptions. We show how asynchronous exceptions easily enable (out of the box) useful programming patterns like speculative execution and some degree of resource management. We prove that programs in MAC$_{async}$ satisfy progress-sensitive non-interference and mechanize our formal claims in the Agda proof assistant.**

I. Introduction

Information-Flow Control [1] (IFC) is a promising approach for preserving confidentiality of data. It tracks how data of different sensitivity levels (e.g., public or sensitive) flows within a program, and raises alarms when confidentiality might be at stake. This technology has been previously used to secure operating systems (e.g., [2, 3]), web browsers (e.g., [4, 5]), and several programming languages (e.g., [6, 7, 8]).

Most language-based approaches for IFC reason about constructions found in programs (e.g., variables, branches, and data structures), while often ignoring aspects of runtime systems which might create *covert channels* (e.g., [9, 10, 11]) capable of producing leaks, e.g., through caches, parallelism, resource usage, etc. To deal with this problem, researchers have proposed security-aware runtime system designs [11, 12]. However, building runtime systems is a major endeavour and these proposals have yet to be implemented. A more lightweight approach to securing runtime systems relies on *special-purpose language constructs* that coordinate the execution of programs with different components of the runtime—e.g., the garbage-collector [10], the scheduler [13], timeouts [14], lazy evaluation [15] and caches [16].[1] While a step in the right direction, designing ad-hoc constructs every time that some coordination with the runtime system is needed feels rather unsatisfactory—an observation that has also been made outside the security arena [17, 18, 19]. In fact, implementing *hooks* in an existing runtime system requires specific knowledge of its internals and considerable expertise. Even worse, the composition of piecemeal security solutions may weaken or even break the security guarantees of the runtime system as a whole. These issues suggest the need for a unified mechanism to close covert channels in the runtime system. As a first step towards this goal, we believe that runtime systems should expose a general *IFC-aware interface* that allow IFC languages to systematically control and secure components of the runtime system. How should programs coordinate with the runtime system through this interface?

In the 70s, Unix-like operating systems conceived *signals* as a limited form of inter process communication (IPC).[2] Signals are no more than asynchronous notifications sent to processes in order to notify them of the occurrence of events, where the origin of signals is either the kernel or other processes. Furthermore, when receiving a signal, process execution can be interrupted during any *non-atomic* instruction—and if the process has previously registered a signal handler, then that routine gets executed. If we think of the kernel as "the runtime" and of processes as our "programs", signals are exactly the mechanism needed to implement the interface that we need! In fact, and generally speaking, the idea of OS-signals have been already internalized by programming languages in the form of *asynchronous exceptions*.

Asynchronous exceptions are raised as a result of external events and can occur at *any point* of the program. As a result, they are considered so difficult to master that many languages (e.g., Python [20] and Java [21]) either restrict or completely forbid programmers from using them. The main reason is that interrupting a program at any point might break, for instance, a data-structure invariant or result in holding a lock indefinitely—and it is not that clear how to get out of such situation.

Despite not being widely adopted in its full expressive power, asynchronous exceptions enable very useful programming patterns: *speculative execution* (i.e., a thread can spawn a child thread and later decide that it does not need the result

[1] In this last case, we are abusing the term runtime to denote "the rest of the system."

[2] https://standards.ieee.org/content/ieee-standards/en/standard/1003_1-2017.html

and kill it), *timeouts*, and *resource management*. We argue that such patterns are desirable to have in any modern IFC system.

Our contributions

In this work, we present **MAC***async*, a Haskell IFC library that extends the concurrent version of **MAC** [22, 23] with asynchronous exception. We formally prove progress-sensitive non-interference (PSNI) [24] for **MAC***async* and provide mechanized proofs in Agda [25] of all our claims as supplementary material to this work. We believe that the extension presented in this paper and its formal security guarantees extend to other Haskell IFC libraries (e.g., **LIO** [26]).

The semantics for asynchronous exceptions in **MAC***async* is inspired by how asynchronous exception are modeled in Haskell [27]—where a mechanism of *masking/unmasking* marks regions of code where asynchronous exceptions can be safely raised. However, allowing untrusted code to mask exceptions *arbitrarily* poses other security risks. For example, a rouge thread could abuse the masking mechanism to exhaust all available computing resources and starve other threads in the system without the risk of being terminated. To avoid that, we propose a fine-grained (selective) masking/unmasking mechanism instead of the traditional *all-or-nothing* approaches, which disable all asynchronous exceptions inside handlers [28, 29]. Furthermore, in contrast with [27], our design forbids raising multiple exceptions at the same time, which, we believe, can too easily disrupt programs in unpredictable ways. While an exception is raised, our language does not raise incoming exceptions, which are, instead, stored in a queue of pending exceptions and raised only when the current one has been handled.

From the security perspective, asynchronous exceptions follow the *no write-down* security check for IFC: when throwing an asynchronous exception, the security level of the source thread should flow to the security level of the recipient. The caveats are, however, in the formalization of masking/unmasking mechanisms and the (progress-sensitive) non-interference proof. For example, it is important for security that asynchronous exception are *deterministically* inserted into the queue of pending exceptions. We utilize *term erasure* as the proof technique and leverage *double-step erasure* to deal with the complexity of our semantics (i.e., concurrency, synchronization variables and asynchronous exceptions), like in previous existing work (e.g., [23, 30, 31]).

In summary, our list of contributions includes:

- An extension to **MAC**, called **MAC***async*, to handle IFC-aware asynchronous exceptions in the presence of concurrency.
- Formal semantics, enforcement, and progress-sensitive non-interference guarantees for **MAC***async*.
- Mechanized proofs of all our claims in approximately 3,000 lines of Agda[3].
- We showcase **MAC***async* and the new programming patterns enabled by asynchronous exceptions with two examples, in which we implement secure versions of (i) a speculative execution combinator, and (ii) a load-balancing controller for sensitive worker threads, respectively.

-- Abstract types
data *Labeled* ℓ τ
data *MAC* ℓ τ

-- Monadic structure for computations
instance *Monad* (*MAC* ℓ)

-- Core operations
label $:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)$
unlabel $:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Labeled}\ \ell_{\mathrm{L}}\ \tau \rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ \tau$

Fig. 1: Core API for **MAC**.

To the best of our knowledge, this work is the first account for asynchronous exceptions in concurrent IFC-systems. The rest of the paper is organized as follows. In Section II, we revisit **MAC**'s API. Section III presents **MAC***async* by example. In Section IV, we extend **MAC**'s semantics to track asynchronous exceptions. In Section V, we introduce asynchronous exceptions and the masking/unmasking mechanisms. Section VI presents our security guarantees. Section VII describes related work and Section VIII concludes.

II. The MAC IFC Library

To help readers get familiar with the **MAC** IFC library [22], we give a brief overview of its API and programming model.

a) Security Lattice: The information flow policies enforced by **MAC** are specified by a security lattice [32], which defines a partial order between security levels (*labels*). These labels represent the sensitivity of program inputs and outputs and the *order* between them dictates which flows of information are allowed in a program. For example, the classic two-point lattice $\mathscr{L} = (\{L, H\}, \sqsubseteq)$ classifies data as either *public* (L) or *secret* (H) and only prohibits sending secret inputs into public outputs, i.e., $H \not\sqsubseteq L$. In **MAC**, the security lattice is embedded in Haskell using standard features of the type-system [22]. In particular, each security label is represented by an abstract datatype and valid flows of information (the $\sqsubseteq$-relation between labels) are encoded using type-class constructs.

b) Security Types: **MAC** enforces security statically by means of special types annotated with security labels. The abstract type *Labeled* ℓ τ associates label ℓ with data of type τ. For example, $\mathit{pwd} :: \mathit{Labeled}\ H\ \mathit{String}$ is a secret string and $\mathit{score} :: \mathit{Labeled}\ L\ \mathit{Int}$ is a public integer. The abstract type *MAC* ℓ τ represents a side-effecful computation that manipulates data labeled with ℓ and whose result has type τ. **MAC** provides a monadic interface to help programmers write secure code. The basic primitives of the interface are *return* and *bind* (written as the infix operator $\ggg$=). Primitive $\mathit{return} :: \tau \rightarrow \mathit{MAC}\ \ell\ \tau$ creates a computation that simply returns a value of type τ without causing side-effects. Primitive $(\ggg\!=) :: \mathit{MAC}\ \ell\ \tau_1 \rightarrow (\tau_1 \rightarrow \mathit{MAC}\ \ell\ \tau_2) \rightarrow \mathit{MAC}\ \ell\ \tau_2$ chains two computations (at the same security level ℓ) together, in a *sequence*. Specifically, program $m \ggg\!= f$ takes the result obtained by executing m and *binds* it to function f, which

[3] Available at https://bitbucket.org/carlostome/mac-async.

$$
\begin{array}{l}
\mathit{fork} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ () \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ () \\
\mathbf{data}\ \mathit{MVar}\ \ell\ \tau \\
\mathit{newMVar} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{MVar}\ \ell_{\mathrm{H}}\ \tau) \\
\mathit{putMVar}\ \ :: \mathit{MVar}\ \ell\ \tau \rightarrow \tau \rightarrow \mathit{MAC}\ \ell\ () \\
\mathit{takeMVar} :: \mathit{MVar}\ \ell\ \tau \rightarrow \mathit{MAC}\ \ell\ \tau
\end{array}
$$

Fig. 2: Concurrent API for **MAC**.

produces the rest of the computation. Our examples use **do**-notation, Haskell syntactic sugar for monadic computations. For instance, we write $\mathbf{do}\ x \leftarrow m; \mathit{return}\ (x+1)$ for the program $m \gg\!= \lambda x \rightarrow \mathit{return}\ (x+1)$, which increments by one the result returned by m.

c) Flows of information: In order to enforce information flow policies, **MAC** regulates the interaction between MAC computations and $\mathit{Labeled}$ data. Computations cannot *write* and *read* labeled data directly, but must use special functions label and $\mathit{unlabel}$ (Fig. 1). These functions create and read labeled data as long as these operations comply with specific security rules, known as *no write-down* and *no read-up* [33]. Intuitively, function label writes some data into a fresh, ℓ_{H}-labeled value as long as the decision to do so depends on less sensitive data, i.e., the computation is labeled with ℓ_{L} such that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. (To help readers, we use subscripts in metavariables ℓ_{L} and ℓ_{H} to indicate that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$). Dually, function $\mathit{unlabel}$ allows ℓ_{H}-labeled computations to read data from lower security levels, i.e., data labeled with ℓ_{L} such that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. In the type-signatures of these functions, the precondition $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ is a type(-class) constraint, which must be statically satisfied when type-checking programs. As a result, programs that attempt to leak secret data, e.g., via *implicit flows*, are ill-typed and rejected by the compiler. In particular, programs cannot branch on *secret* H-labeled data directly, but must use $\mathit{unlabel}$ first to extract its content. Once unlabeled, secret data can only be manipulated within a computation labeled with H thanks to the type of $\mathit{unlabel}$ and *bind*. Then, trying to use function label to create *public* L-labeled data triggers a type error that represents a violation of the no write-down rule. Specifically, an attempt to create public data from within a secret context generates an *unsatisfiable* type constraint $H \sqsubseteq L$, arising from the use of label.

MAC incorporates other kinds of resources (e.g., references and network sockets) in a similar way. Resources are encapsulated in *labeled* resources handlers and the API exposed to labeled computations is designed so that the read and write side-effects of each operation respect the *no read-up* and *no write-down* rules.

d) Concurrency: Extending IFC languages with concurrency is a delicate task because threads provide attackers with new means to leak data. For example, the possibility of executing computations concurrently magnifies the bandwidth of the termination covert channel [34]. This channel enables brute force attacks in which threads try to guess the secret and enter into a loop to suppress public outputs if they succeed. Even worse, the combination of concurrency and shared resources can introduce subtle *internal-timing channels* [35]. This covert channel is exploited by attacks that influence the (public) outcome of *data races* with secret data [9, 34]. To support concurrency securely, **MAC**: (i) decouples computations that manipulate secret data from computations that can generate public outputs, and (ii) prevents threads (labeled computations) from affecting data races between threads at lower security levels. Primitive fork (Fig. 2) allows a ℓ_{L}-labeled computation to fork a thread at a higher security level, i.e., labeled with ℓ_{H} such that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. Intuitively, forking constitutes a *write* operation and thus the type of fork enforces the no write-down rule. **MAC** does not implements threads directly, but relies on Haskell *green* (lightweight user-level) threads. These threads are managed by the GHC runtime system running a round-robin scheduler, which is compatible with the security guarantees of **MAC** [23, 30].

e) Synchronization Variables: **MAC** supports shared mutable state in the form of synchronization variables, following the style of Concurrent Haskell [36]. The abstract type $\mathit{MVar}\ \ell\ \tau$ (Fig. 2) represents a synchronization variable that can be either *empty* or *full* with a value of type τ at security level ℓ. Threads can create and atomically access synchronization variables with functions $\mathit{newMVar}$, $\mathit{putMVar}$ and $\mathit{takeMVar}$. Function $\mathit{newMVar}$ creates a synchronization variable initially full with the given value. (Like label and fork, function $\mathit{newMVar}$ performs a write side-effect, thus its type signature has a similar security check). Functions $\mathit{putMVar}$ and $\mathit{takeMVar}$ allow threads to write and read shared variables *synchronously*. In particular, these functions *block* threads trying to read or write variables in the wrong "state". For example, function $\mathit{putMVar}$ writes a value into an *empty* variable and blocks the thread if the variable is full. Dually, function $\mathit{takeMVar}$ empties a *full* variable and returns its content and blocks the caller otherwise. Notice that $\mathit{putMVar}$ and $\mathit{takeMVar}$ perform both *read* and *write* side-effects: they must always read the variable to determine whether the caller should be blocked. Then, the *no read-up* and *no write-down* security rules imply that these functions are secure only when they operate within the same security level, i.e., both the variable and the computation are labeled with ℓ [22].

III. MAC$_{async}$ BY EXAMPLE

Asynchronous exceptions enable useful programming patterns that, to our knowledge, cannot be coded securely in any existing IFC language. We illustrate some of these idioms in **MAC**$_{async}$, which extends **MAC** with three new primitives $\mathit{throwTo}$, mask (unmask), and catch. These primitives allow threads to (1) send signals to threads at higher security levels by throwing exceptions *asynchronously*, (2) *suppress* (*enable*) exceptions in specific regions of code, and (3) *react* to exceptions by running their corresponding exception handler, respectively.

Example 1 (Speculative execution)**.** Imagine two implementations of the same algorithm whose performance depends on the input. Instead of settling for one, we could run both concurrently and just return the output of the first that finishes. At that point the thread computing the other algorithm may be killed since its result is no longer necessary. We can implement such a combinator for speculative execution in **MAC**$_{async}$ using asynchronous exceptions. First, we declare $Kill :: Exception$ as a new exception, and define $kill\ t$, a function that sends exception $Kill$ asynchronously to the thread identified by t.[4]

```
1  data Exception = Kill  | ...
2  kill t = throwTo t Kill
```

Then, we define the combinator *speculate*, which receives two computations c_1 and c_2 to run speculatively.

```
3  speculate :: MAC ℓ a → MAC ℓ a → MAC ℓ a
4  speculate c₁ c₂ = do
5     m ← newEmptyMVar
6     t₁ ← fork (c₁ ≫= putMVar m)
7     t₂ ← fork (c₂ ≫= putMVar m)
8     r  ← takeMVar m
9     kill t₁; kill t₂
10    return r
```

The combinator creates an *empty* synchronization variable m (line 5) and forks two threads (6–7), which run computations c_1 and c_2 concurrently and then write the result to m. When the combinator reads variable m (8), it *blocks* until either thread terminates and fills it with the result. When this happens, the combinator resumes, kills the children threads (one may still be running) (9), and returns the result (10).

Example 2 (Thread pool)**.** This example presents the code of a *controller* thread that maintains a pool of *worker* threads to perform computations on a stream of incoming (sensitive) inputs. In this scheme, the controller thread manages the worker threads in the pool by reacting to asynchronous exceptions sent by other (public and secret) threads in the system. For example, when some secret input becomes available, a thread can send an exception $Input_H\ secret$ to the controller thread, which extracts the secret data and forwards it to the first available worker thread to process it. Similarly, when the thread pool is no longer needed, it can be deallocated by sending the exception $Kill$ to the controller, which then kills each worker thread in the pool. In the same way, the controller could be programmed to react to specific exceptions and carry out even more tasks (e.g., dynamically resizing the thread pool).

To set up this scheme, a thread calls function *initTP* (Fig. 3) to initialize the thread pool and start the controller thread. Function $initTP\ \ n\ \ f$ allocates an empty synchronization variable m (line 6), forks a pool of n worker threads executing function f (line 7), collects their identifiers ts, and passes it to the controller thread (line 8). As new input becomes available, the controller writes it to the shared variable m, which is then read by one of the workers and its content processed via function f (line 11). To avoid getting killed in the middle of a computation, worker threads *mask* exception $Kill$ while processing data, thus ensuring that they always complete ongoing computations without aborting prematurely. It may seem erroneous to mask also instruction *takeMVar*: can this cause a worker thread to *block indefinitely* waiting for new input? No, in Concurrent Haskell, and **MAC**$_{async}$, operations that can block indefinitely (like *takeMVar*) are *interruptible*, i.e., they can receive and raise asynchronous exceptions even in masked blocks [27].

```
1  type Data = ...
2  data Exception = Kill | Input_ℓ (Labeled ℓ Data) | ...
3  type Size = Int
4  initTP :: Size → (Data → MAC H ()) → MAC L (TId H)
5  initTP n f = do
6     m ← newEmptyMVar
7     ts ← forM [1 . . n] (λ_ → fork (worker f m))
8     fork (controller ts m)
9  worker :: (Data → MAC H ()) → MVar H Data → MAC H ()
10 worker f m = do
11    mask [Kill] (takeMVar m ≫= f)
12    worker f m
13 controller :: [TId H] → MVar H Data → MAC H ()
14 controller ts m =
15    let wait = newEmptyMVar ≫= takeMVar in
16    catch wait
17      [(Input_H secret,
18        mask [Input_H, Kill]
19             (do s ← unlabel secret
20                 putMVar m s
21                 unmask [Input_H, Kill] (controller ts m)))
22      , (Kill,
23        mask [Input_H, Kill] (forM ts kill)) ]
```

Fig. 3: Thread pool example.

Function $controller\ \ ts\ \ m$ implements a controller thread for the thread pool ts sharing variable m. As long as it receives no exception, the controller thread simply waits on an always-empty synchronization variable via *wait* (line 15). When the thread receives an exception, it resumes and executes the corresponding code in the list of exception handlers. In particular, when new secret input becomes available ($Input_H\ secret$), it unlabels the secret (line 19) and writes it to variable m (line 20), so that the worker threads can process it. Notice that if variable m is *full* at this point, then some previous input is still waiting to be processed (all workers threads are busy) and the controller just waits on the variable. As soon as a worker thread completes, it empties the variable containing the pending input, and the controller resumes by writing the variable; then it continues to wait for further exceptions. To avoid dropping any input, the controller thread masks exceptions $Kill$ and $Input_H$ (line 18) while processing requests. For example, if exceptions were not properly masked

[4] In **MAC**$_{async}$, primitive *fork* returns the identifier of the child thread to the parent.

in that block of code, the controller could receive an exception, e.g., *Kill*, which would terminate the thread while trying to feed the last input received to the workers. Once done, the controller unmasks the exceptions again (line 21) and continues to wait for new input. After receiving and *eventually* raising the exception *Kill*, the controller thread propagates it to all the workers in the pool (line 23) and then terminates. Also in this case, the controller thread masks the other exceptions, which could otherwise prematurely terminate the controller and leave some of the worker threads alive.

The example, however, has a catch! Primitive *putMVar* may also block the controller indefinitely like *takeMVar*, and thus may likewise be *interrupted* and raise an exception, even if that exception is masked. As a result, the controller thread could also be interrupted on line 20 and drop the current input. To fix the program, we introduce the combinator *retry killed m ss*, which repeatedly attempts to fill variable m with the inputs pending in list ss while handling other exceptions.

```
24 retry :: Bool → [TId H] → MVar H Data
25                               → [Data] → MAC H ()
26 retry killed ts m [] =
27   if killed then forM ts kill; exit else return ()
28 retry killed m (s : ss) =
29   catch (putMVar m s)
30          [(Input_H secret,
31            do s' ← unlabel secret
32               if killed
33                  then retry killed ts m (s : ss)
34                  else retry killed ts m ((s : ss) ++ [s'])
35          , (Kill, retry True ts m (s : ss)) ]
```

If further inputs are received while executing *retry*, the function appends them to list *ss* to avoid dropping them, and thus ensuring that they will eventually be delivered to the workers. If the controller receives exception *Kill* while retrying, the boolean flag *killed* is switched on and further inputs are discarded. In this case, when all the inputs received before *Kill* are dispatched, the controller kills the worker threads and terminates with *exit* (line 27)—the function *retry* assumes exceptions *Input H* and *Kill* are masked so this operation will not be interrupted. In conclusion, to repair the code of *controller*, we simply replace *putMVar m s* (line 20) with *retry False ts m* $[s]$.

Even though relatively simple, these examples cannot be coded in IFC languages without support for asynchronous communication like **MAC**. In these languages, synchronous primitives (e.g., *MVar*) must be restricted to operate within a single security level for security reasons, as explained in Section II and [23]. For instance, if only synchronous communication was available, then the *controller* thread from our second example could not receive commands from public (L-labeled) threads.[5]

Types: $\tau ::= () \mid \mathit{Bool} \mid \tau_1 \to \tau_2$
Values: $v ::= () \mid \mathit{True} \mid \mathit{False} \mid \lambda x.t$
Terms: $t ::= t_1\ t_2 \mid \mathbf{if}\ t\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \mid v$

Fig. 4: Core syntax.

Types: $\tau ::= \cdots \mid \mathit{MAC}\ \ell\ \tau \mid \mathit{Labeled}\ \ell\ \tau$
Values: $v ::= \cdots \mid \mathit{Labeled}\ t \mid \mathit{return}\ t$
Terms: $t ::= \cdots \mid \mathit{label}\ t \mid \mathit{unlabel}\ t \mid t_1 \gg\!= t_2$

(UNLABEL$_1$)
$$\frac{t_1 \rightsquigarrow t_2}{\mathit{unlabel}\ t_1 \longrightarrow \mathit{unlabel}\ t_2}$$

(UNLABEL$_2$)
$$\mathit{unlabel}\ (\mathit{Labeled}\ t) \longrightarrow \mathit{return}\ t$$

Fig. 5: Syntax and semantics of **MAC***async* (excerpts).

IV. Formal Semantics

A. Core of **MAC***async*

From a security perspective, the interaction between synchronization variables, asynchronous exceptions, and exception masking is a delicate matter. **MAC***async* implements these primitives on top of those provided by Concurrent Haskell, whose runtime is not designed with security in mind. For example, the fact that a thread may be able to resume another by sending an asynchronous exception [27] (as explained in the second example above) may introduce subtle internal timing covert channels that weaken the security guarantees of **MAC**. To rule that out, we extend the small-step semantics of **MAC** from [23] with asynchronous exceptions and perform a rigorous, comprehensive security analysis of the whole language.

The core of **MAC***async* is the standard call-by-name λ-calculus with boolean and unit type (Fig. 4). We specify the side-effect free semantics of the core λ-calculus (e.g., function abstraction, application) as a small-step reduction relation, $t_1 \rightsquigarrow t_2$, which denotes that term t_1 reduces in one step to t_2. These reduction rules are standard and we completely omit them in this presentation.

The defining feature of **MAC***async* is the security monad *MAC*, which encapsulates computations that may produce side-effects. Figure 5 specifies the syntax and part of the semantics for the side-effectful constructs of the language. The small-step relation $t_1 \longrightarrow t_2$ denotes a single *sequential* step that brings term t_1 of type $\mathit{MAC}\ \ell\ \tau$ to t_2. Rule (UNLABEL$_1$) reduces term $\mathit{unlabel}\ t_1$ to $\mathit{unlabel}\ t_2$ by evaluating the argument through a *pure* semantics step $t_1 \rightsquigarrow t_2$. When the argument is evaluated, rule (UNLABEL$_2$) extracts the content of the labeled value and returns it in the security monad.

B. Synchronization Variables

Figure 6 extends **MAC***async* with synchronization variables. The store Σ is partitioned by label into separate memory segments S, each consisting of a list of memory cells c, which can be either *full* with a term $(\!|\, t \,|\!)$ or *empty* ($\otimes$). A value

[5] In **MAC**, a public thread could technically communicate asynchronously with a secret thread by updating a secret, mutable reference. However, these labeled references would inevitably introduce serious data races and thus do not represent a viable alternative.

Store: $\Sigma \in \mathit{Label} \rightarrow \mathit{Memory}$

Memory: $S ::= [\,] \mid c : S$

Cell: $c ::= \otimes \mid (\!|\, t \,|\!)$

Addresses: $n \in \mathbb{N}$

Types: $\tau ::= \cdots \mid \mathit{MVar}\ \ell\ \tau$

Values: $v ::= \cdots \mid \mathit{MVar}_{\ell}\ n$

Terms: $t ::= \cdots \mid \mathit{newMVar}_{\ell}\ t \mid \mathit{takeMVar}\ t \mid \mathit{putMVar}\ t_1\ t_2$

(New)
$$\frac{n = |\Sigma(\ell)|}{\Sigma, \mathit{newMVar}_{\ell}\ t \longrightarrow \Sigma[(\ell, n) \mapsto (\!|\, t \,|\!)], \mathit{return}\ (\mathit{MVar}_{\ell}\ n)}$$

(Put$_1$)
$$\frac{t_1 \rightsquigarrow t_1'}{\Sigma, \mathit{putMVar}\ t_1\ t_2 \longrightarrow \Sigma, \mathit{putMVar}\ t_1'\ t_2}$$

(Put$_2$)
$$\frac{(\ell, n) \mapsto \otimes \in \Sigma \qquad \Sigma' = \Sigma[(\ell, n) \mapsto (\!|\, t \,|\!)]}{\Sigma, \mathit{putMVar}\ (\mathit{MVar}_{\ell}\ n)\ t \longrightarrow \Sigma', \mathit{return}\ ()}$$

Fig. 6: Syntax and semantics for synchronization variables.

$\mathit{MVar}_{\ell}\ n$ denotes a synchronization variable that refers to the n-th cell of the ℓ-labeled memory segment in the store.[6]

In rule (New), primitive $\mathit{newMVar}_{\ell}\ t$ allocates a new memory cell containing term t in the ℓ-labeled segment of the store, at *fresh* address $n = |\Sigma(\ell)|$, i.e., $\Sigma[(\ell, n) \mapsto (\!|\, t \,|\!)]$, and returns the corresponding synchronization variable $\mathit{MVar}_{\ell}\ n$. Term $\mathit{putMVar}_{\ell}\ t_1\ t_2$ writes term t_2 into the *empty* cell pointed by the synchronization variable t_1. To do that, rule (Put$_1$) starts evaluating the variable t_1 through a pure semantics step $t_1 \rightsquigarrow t_1'$. When the variable is fully evaluated, e.g., $\mathit{MVar}_{\ell}\ n$, rule (Put$_2$) takes over and writes the given term t in the cell identified by (ℓ, n), i.e., $\Sigma[(\ell, n) \mapsto (\!|\, t \,|\!)]$. Notice that the term steps only if the cell in the store Σ is initially *empty*, i.e., $(\ell, n) \mapsto \otimes \in \Sigma$. If the cell is *full*, the term cannot be reduced by any other rule of the semantics and gets *stuck*, capturing the intended *blocking* behavior of synchronization variables. We omit the rules for $\mathit{takeMVar}$, which follow a similar pattern [23].

C. Concurrency

Unlike previous concurrent incarnations of **MAC**, threads in **MAC**$_{\mathit{async}}$ can communicate with each other by sending signals in the form of asynchronous exceptions. To enable this form of communication, the runtime system assigns a unique thread identifier to each thread of the system. Thread identifiers are *opaque* to avoid leaking secret data through the number of threads in the system, and *labeled* to prevent sensitive threads from sending exceptions to threads at lower security levels. **MAC**$_{\mathit{async}}$ incorporates thread identifiers with values $\mathit{TId}_{\ell}\ n$ of the new primitive type $\mathit{TId}\ \ell$, whose label ℓ represents the static security level of the thread identified by n. Thread identifiers are also *unforgeable* and only generated automatically by the runtime system each time a new thread is forked.

Events: $e ::= \mathbf{step} \mid \mathbf{fork}_{\ell}(t)$

Thread Id: $n \in \mathbb{N}$

Thread Id Map $\phi \in \mathit{Label} \rightarrow \mathit{Thread\ Id}$

Types: $\tau ::= \cdots \mid \mathit{TId}\ \ell$

Values: $v ::= \cdots \mid \mathit{TId}_{\ell}\ n$

Terms: $t ::= \cdots \mid \mathit{fork}_{\ell}\ t$

(Fork)
$$\frac{n = \phi(\ell)}{\Sigma, \mathit{fork}_{\ell}\ t \xrightarrow{\mathbf{fork}_{\ell}(t)}_{\phi} \Sigma, \mathit{return}\ (\mathit{TId}_{\ell}\ n)}$$

Fig. 7: Syntax and semantics of fork.

$$\mathit{fork} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ () \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{TId}\ \ell_{\mathrm{H}})$$

Figure 7 extends the sequential calculus of **MAC**$_{\mathit{async}}$ with concurrency primitives. To simplify our security analysis, term $\mathit{fork}_{\ell}\ t$ is annotated with the security label ℓ of thread t of type $\mathit{MAC}\ \ell\ ()$. Similarly to [23], we decorate the sequential reduction relation from above with *events*, which inform the top-level scheduler of the execution of sequential commands that have global effects. For example, event $\mathbf{fork}_{\ell}(t)$ indicates that thread t at security level ℓ has been forked and event $\mathbf{step}$ denotes an uninteresting (*silent*) sequential step. Later, we extend the category of events to keep track of asynchronous exceptions as well. Sequential steps are also parameterized by a *thread id map* ϕ, which represents a source of fresh thread identifiers for each security level. The use of this map is exemplified by rule (Fork). Whenever a new thread is forked, e.g., $\mathit{fork}_{\ell}\ t$, we use the label annotation ℓ to generate a fresh identifier $n = \phi(\ell)$, which is then returned in the monad wrapped in the constructor of thread identifiers, i.e., $\mathit{TId}_{\ell}\ n$.

Figure 8 introduces the top-level semantics relation that formalizes how concurrent configurations evolve. Concurrent configurations are pairs $\langle \Sigma, \Theta \rangle$ consisting of the concurrent store Σ and a map of thread pools Θ. The thread pool map Θ maps each label of the lattice to the list of threads T_s at that security level, currently in the system. Each rule of the concurrent semantics constructs the source of fresh thread identifiers ϕ from the thread pool map Θ of the initial configuration by means of the function $\mathsf{nextId}(\Theta) = \lambda\ell.|\Theta(\ell)|$.

A concurrent step $\ell, n \vdash c_1 \hookrightarrow c_2$ indicates that configuration c_1 steps to c_2, while running the thread identified (ℓ, n), i.e., the n-th thread of the ℓ-labeled thread pool. The particular scheduler used to determine which thread runs at every step is not very relevant for our discussion, therefore we omit it in our semantics. It suffices to say that the security guarantees of **MAC**$_{\mathit{async}}$ carry over for a wide range of deterministic schedulers [23] (as witnessed by our mechanized proofs) and include the Round Robin scheduler adopted in Concurrent Haskell. The concurrent rules rely on sequential events to determine which step to take. For example, rule (Seq) extracts the running thread from the thread pool, i.e., $\Theta[(\ell, n) \mapsto t_1]$, which steps *silently*, i.e., generating event

[6] Some terms in the calculus carry a label annotation that is inferred from its type. For example, the label ℓ in $\mathit{MVar}_{\ell}\ n$ comes from its type $\mathit{MVar}\ \ell\ \tau$.

Configuration: $C ::= \langle \Sigma, \Theta \rangle$
Thread Pool Map: $\Theta \in \mathit{Label} \to \mathit{Thread\ Pool}$
Thread Pool: $T_s ::= [\,] \mid (th, T_s)$
Thread State: $th ::= t$

(SEQ)

$$\frac{\phi = \mathsf{nextId}(\Theta) \qquad \Sigma_1, t_1 \xrightarrow{\textbf{step}}_{\phi} \Sigma_2, t_2}{\ell, n \vdash \langle \Sigma_1, \Theta[(\ell, n) \mapsto t_1] \rangle \hookrightarrow \langle \Sigma_2, \Theta[(\ell, n) \mapsto t_2] \rangle}$$

(FORK)

$$\frac{\begin{array}{c} \phi = \mathsf{nextId}(\Theta_1) \qquad n' = \phi(\ell') \\ \Theta_2 = \Theta_1[(\ell, n) \mapsto t_2] \qquad \Sigma, t_1 \xrightarrow{\textbf{fork}_{\ell'}(t)}_{\phi} \Sigma, t_2 \end{array}}{\ell, n \vdash \langle \Sigma, \Theta_1[(\ell, n) \mapsto t_1] \rangle \hookrightarrow \langle \Sigma, \Theta_2[(\ell', n') \mapsto t] \rangle}$$

Fig. 8: Syntax and semantics of concurrent **MAC**$_{async}$.

$$\begin{array}{ll} \mathit{throwTo} & :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{TId}\ \ell_{\mathrm{H}} \to \chi \to \mathit{MAC}\ \ell_{\mathrm{L}}\ () \\ \mathit{catch} & :: \mathit{MAC}\ \ell\ \tau \to [(\chi, \mathit{MAC}\ \ell\ \tau)] \to \mathit{MAC}\ \ell\ \tau \end{array}$$

Fig. 9: **MAC**$_{async}$ API for asynchronous exceptions.

step, and thus the rule only reinserts the thread term in the thread pool, i.e., $\Theta[(\ell, n) \mapsto t_2]$. In contrast, event $\textbf{fork}_{\ell'}(t)$ in rule (FORK) indicates that the running thread has forked, therefore the rule reinserts the parent thread in the pool, i.e., $\Theta_2 = \Theta_1[(\ell, n) \mapsto t_2]$, and also adds its child at the corresponding security level ℓ' and fresh index $n' = \phi(\ell')$, i.e., $\Theta_2[(\ell', n') \mapsto t]$.

V. ASYNCHRONOUS EXCEPTIONS

MAC$_{async}$ supports sending and handling asynchronous exceptions by means of two new primitives *throwTo* and *catch*, see Figure 9. Primitive $\mathit{throwTo}\ t\ \xi$ raises the exception ξ of abstract type χ *asynchronously* in the thread with identifier t. Intuitively, this operation constitutes a write effect, therefore **MAC**$_{async}$ restricts its API according to the *no write-down* rule to enforce security. To this end, the API ensures that the security label of the receiver thread (ℓ_{H}) is at least as sensitive as the label of the sender (ℓ_{L}) through the type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. Once delivered and raised, asynchronous exceptions behave like synchronous exceptions. They disrupt the execution of the receiving thread in the usual way, with the exception bubbling up in the code of the thread and, if uncaught, eventually crashing it. Threads can recover from exceptions by wrapping regions of code in a *catch* block. The same mechanism, allows threads to *react* to asynchronous signals by handling exceptions appropriately. Primitive $\mathit{catch}\ t\ \mathit{hs}$ takes as a parameter a computation t and a list containing pairs of exceptions and handlers. Then, if an exception ξ is raised during the execution of t, the handler corresponding to the first exception *matching* ξ in the list hs, if there is one, gets executed.

Figure 10 extends the calculus with value $\mathit{raise}\ \xi$, which indicates that the computation is in an *exceptional state*, and a new event $\textbf{throw}_{\ell}(\xi, n)$, which instructs the runtime to deliver exception ξ to the thread identified by (ℓ, n). To model how asynchronous exceptions propagate precisely, we add new rules both to the sequential and concurrent semantics. Rule (THROWTO$_1$) evaluates the thread identifier in term $\mathit{throwTo}\ t_1\ \xi$, which reduces to $\mathit{throwTo}\ t_2\ \xi$ through the pure step $t_1 \rightsquigarrow t_2$. (For simplicity, our model assumes that exceptions are already evaluated in terms, thus the rules do not need to reduce them). When the thread identifier is fully evaluated, i.e., it is of the form $\mathit{TId}_{\ell}\ n$, rule (THROWTO$_2$) generates event $\textbf{throw}_{\ell}(\xi, n)$ and returns unit. The rule reflects the *non-blocking* behavior of *throwTo*, which always succeeds as soon as the thread identifier is evaluated and regardless of the state of the receiving thread. This design decision has important security implications that we discuss further in Section V-C. Rule (CATCH$_1$) executes the computation t_1 in term $\mathit{catch}\ t_1\ \mathit{hs}$. If during the execution of t_1 the computation receives some exception ξ, and the exception propagates up to the exception handler, then the term reduces to $\mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs}$ and rules (CATCH$_{\xi 1}$) and (CATCH$_{\xi 2}$) determine whether the exception gets handled or not. In these rules, function $\mathsf{first}(\mathit{hs}, \xi)$ searches for a handler corresponding to exception ξ in the list hs. To do so, the function traverses the list of exception-handler pairs hs left-to-right until it finds a pair whose left component is equal to exception ξ. If a handler for that exception is in the list, i.e., $h = \mathsf{first}(\mathit{hs}, \xi)$, then (CATCH$_{\xi 1}$) passes control to it. If no handler matches the exception, i.e., $\emptyset = \mathsf{first}(\mathit{hs}, \xi)$, then rule (CATCH$_{\xi 2}$) simply propagates the exception.

A. Masking Exceptions

Asynchronous exceptions are typically sent in response to external events such as user interrupts and exceeding resource limits. These exceptions can disrupt threads unpredictably, at any moment during their execution, and end up breaking code invariants and leaving shared data structures in an inconsistent state. For example, an incoming exception may crash a thread inside a critical section and cause it to hold a lock indefinitely, without the possibility of cleaning up. Therefore, writing robust code in the presence of asynchronous exceptions requires a mechanism to *temporarily* suppress exceptions in critical sections that must not be interrupted. Inspired by [27], **MAC**$_{async}$ sports two *scoped* combinators, *mask* and *unamsk*, to disable and enable specific exceptions in a given code region, respectively.[7]

$$\begin{array}{ll} \mathit{mask} & :: \chi \to \mathit{MAC}\ \ell\ \tau \to \mathit{MAC}\ \ell\ \tau \\ \mathit{unmask} & :: \chi \to \mathit{MAC}\ \ell\ \tau \to \mathit{MAC}\ \ell\ \tau \end{array}$$

Intuitively, primitive $\mathit{mask}\ \xi\ t$ runs computation t with exceptions ξ disabled. If such an exception is received during the execution of t, the exception is *not* dropped, but stored in a buffer of pending exceptions ξ_s and raised once the execution goes past the *mask* instruction. Term $\mathit{unmask}\ \xi\ t$

[7] Even though these primitives take only a single exception as an argument, they are equivalent to the multi-exception variants used in Section III, i.e., $\mathit{mask}\ [\xi_1, \xi_2]\ t$ behaves exactly like $\mathit{mask}\ \xi_1\ (\mathit{mask}\ \xi_2\ t)$.

$$\begin{array}{lll}
\text{Exceptions:} & \xi & \\
\text{Events:} & e\ ::= \cdots & \mid\ \textbf{throw}_\ell(\xi, n) \\
\text{Values:} & v\ ::= \cdots & \mid\ \mathit{raise}\ \xi \\
\text{Handlers:} & \mathit{hs} ::= [\,] & \mid\ (\xi, t) : \mathit{hs} \\
\text{Terms:} & t\ ::= \cdots & \mid\ \mathit{throwTo}\ t\ \xi\ \mid\ \mathit{catch}\ t\ \mathit{hs}
\end{array}$$

(ThrowTo$_1$)

$$\frac{t_1 \rightsquigarrow t_2}{\Sigma, \mathit{throwTo}\ t_1\ \xi \xrightarrow{\textbf{step}}_\phi \Sigma, \mathit{throwTo}\ t_2\ \xi}$$

(ThrowTo$_2$)

$$\frac{}{\Sigma, \mathit{throwTo}\ (\mathit{TId}_\ell\ n)\ \xi \xrightarrow{\textbf{throw}_\ell(\xi,n)}_\phi \Sigma, \mathit{return}\ ()}$$

(Catch$_1$)

$$\frac{\Sigma_1, t_1 \xrightarrow{e}_\phi \Sigma_2, t_2}{\Sigma_1, \mathit{catch}\ t_1\ \mathit{hs} \xrightarrow{e}_\phi \Sigma_2, \mathit{catch}\ t_2\ \mathit{hs}}$$

(Catch$_{\xi 1}$)

$$\frac{h = \mathsf{first}(\mathit{hs}, \xi)}{\Sigma, \mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs} \xrightarrow{\textbf{step}}_\phi \Sigma, h}$$

(Catch$_{\xi 2}$)

$$\frac{\emptyset = \mathsf{first}(\mathit{hs}, \xi)}{\Sigma, \mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs} \xrightarrow{\textbf{step}}_\phi \Sigma, \mathit{raise}\ \xi}$$

Fig. 10: Syntax and semantics for asynchronous exceptions.

works the other way around and enables exceptions ξ while executing t. In general, whether an exception received by a thread should be raised immediately or temporarily suppressed depends on the *masking context* of the thread. Intuitively, the masking context at each execution point depends on the (nested) *mask* and *unmask* instructions crossed up to that point. For instance, if program $\mathit{unmask}\ \xi\ (\mathit{mask}\ \xi'\ t)$ receives exception ξ while executing t, and if $\xi \not\equiv \xi'$ and t does not contain any $\mathit{mask}\ \xi$ instruction, then the exception gets raised, i.e., $\mathit{unmask}\ \xi\ (\mathit{mask}\ \xi'\ (\cdots\ \mathit{raise}\ \xi\ \cdots))$.

Figure 11 presents the sequential semantics of *mask* and *unmask*. The masking context M is a map from exceptions to booleans, representing a bit vector that indicates which exceptions can be raised in the reduction steps. To keep track of exceptions, the sequential relation carries the list of pending exceptions ξ_s, on the left, and the list of remaining exceptions ξ'_s on the right of the arrow. Further, the arrow is annotated with the masking context of the thread (M). Rules (Mask) and (Unmask) modify the masking context accordingly via functions $\mathsf{mask}(M, \xi) = \lambda \xi'.\xi \equiv \xi' \ \vee\ M(\xi')$ and $\mathsf{unmask}(M, \xi)$ (analogous). In particular, the rules reduce term $\mathit{mask}\ \xi\ t$ (respectively $\mathit{unmask}\ \xi\ t$) by executing term t with modified mask M_2 obtained from disabling (enabling) exception ξ in the current mask M_1, i.e., $M_2 = \mathsf{mask}(M_1, \xi)$ ($M_2 = \mathsf{unmask}(M_1, \xi)$). When a nested, *masked* computation has completed, either successfully ($\mathit{return}\ t$) or not ($\mathit{raise}\ \xi'$), rules (Mask$_1$) and (Mask$_\xi$) simply propagate the result.

$$\begin{array}{lll}
\text{Mask:} & M \ \in\ \chi \to \mathit{Bool} & \\
\text{Terms:} & t\ ::= \cdots & \mid\ \mathit{mask}\ \xi\ t\ \mid\ \mathit{unmask}\ \xi\ t \\
\text{Exception list:} & \xi_s\ ::= [\,]\ \mid\ (\xi : \xi_s) &
\end{array}$$

(Mask)

$$\frac{M_2 = \mathsf{mask}(M_1, \xi) \qquad \Sigma_1, t_1, \xi_s \xrightarrow{e}_{(\phi, M_2)} \Sigma_2, t_2\ \xi'_s}{\Sigma_1, \mathit{mask}\ \xi\ t_1, \xi_s \xrightarrow{e}_{(\phi, M_1)} \Sigma_2, \mathit{mask}\ \xi\ t_2, \xi'_s}$$

(Unmask)

$$\frac{M_2 = \mathsf{unmask}(M_1, \xi) \qquad \Sigma_1, t_1, \xi_s \xrightarrow{e}_{(\phi, M_2)} \Sigma_2, t_2, \xi'_s}{\Sigma_1, \mathit{unmask}\ \xi\ t_1, \xi_s \xrightarrow{e}_{(\phi, M_1)} \Sigma_2, \mathit{unmask}\ \xi\ t_2, \xi'_s}$$

(Mask$_1$)

$$\Sigma, \mathit{mask}\ \xi\ (\mathit{return}\ t), \xi_s \xrightarrow{\textbf{step}}_{(\phi, M)} \Sigma, \mathit{return}\ t, \xi_s$$

(Mask$_\xi$)

$$\Sigma, \mathit{mask}\ \xi\ (\mathit{raise}\ \xi'), \xi_s \xrightarrow{\textbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi', \xi_s$$

Fig. 11: Syntax and semantics of *mask* (*unmask* is similar).

(Bind-Raise)

$$\frac{\mathsf{unmasked}_{[\,]}(M, \xi_s) = (\xi, \xi'_s)}{\Sigma, \mathit{return}\ t_1 \mathbin{>\!\!>\!\!=} t_2, \xi_s \xrightarrow{\textbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi, \xi'_s}$$

(Bind$_2$)

$$\frac{\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset}{\Sigma, \mathit{return}\ t_1 \mathbin{>\!\!>\!\!=} t_2, \xi_s \xrightarrow{\textbf{step}}_{(\phi, M)} \Sigma, t_2\ t_1, \xi_s}$$

(Bind$_\xi$)

$$\Sigma, \mathit{raise}\ \xi \mathbin{>\!\!>\!\!=} t, \xi_s \xrightarrow{\textbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi, \xi_s$$

Fig. 12: *Masking* semantics of bind ($\mathbin{>\!\!>\!\!=}$).

The masking context M and the list of pending exceptions ξ_s determine whether any exception in the list should be raised or not. To reflect that, we need to adapt the semantics rules for most constructs of the calculus. Figure 12 shows the modifications for the monadic bind ($\mathbin{>\!\!>\!\!=}$). (The rules for the other constructs are modified in a similar way, we refer the reader to the Agda mechanization for details).

First, we define function $\mathsf{unmasked}_{\xi'_s}(M, \xi_s)$, which extracts from the list of pending exceptions ξ_s the *first* exception ξ that is *unmasked* in M, i.e., such that $\neg\ M(\xi)$, The function walks down the list recursively and accumulates the exceptions ξ that are *masked* in M, i.e., such that $M(\xi)$, in the list ξ'_s, which is then returned together with the rest of the list ξ_s, when an unmasked exception is found. If all the exceptions in the list are masked, the function simply returns $\emptyset$.

$$\mathsf{unmasked}_{\xi'_s}(M, \xi_s) = \begin{cases} \emptyset & \text{if } \xi_s = [\,] \\ (\xi, \xi'_s \mathbin{+\!\!+} \xi''_s) & \text{if } \xi_s = \xi : \xi''_s \text{ and } \neg\ M(\xi) \\ \mathsf{unmasked}_{(\xi'_s \mathbin{+\!\!+} [\xi])}(M, \xi''_s) & \text{if } \xi_s = \xi : \xi''_s \text{ and } M(\xi) \end{cases}$$

$$(\textsc{Take}_1)\quad \frac{(\ell, n) \mapsto (\!| \, t \, |\!) \in \Sigma \qquad \mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset \qquad \Sigma' = \Sigma[(\ell, n) \mapsto \otimes]}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_\ell\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma', \mathit{return}\ t, \xi_s}$$

$$(\textsc{Take-Raise-Unmasked})\quad \frac{(\ell, n) \mapsto c \in \Sigma \qquad \mathsf{unmasked}_{[\,]}(M, \xi_s) = (\xi, \xi_s')}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_\ell\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi, \xi_s'}$$

$$(\textsc{Take-Raise-Masked})\quad \frac{(\ell, n) \mapsto \otimes \in \Sigma \qquad \mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset \qquad \xi_s = \xi : \xi_s'}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_\ell\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi, \xi_s'}$$

Fig. 13: Synchronization variables and exceptions.

In the *elimination* rules of the semantics, we apply function $\mathsf{unmasked}_{[\,]}(M, \xi_s)$ to determine whether a pending exception should be raised. For example, if all exceptions are masked, i.e., $\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset$, then rule ($\textsc{Bind}_2$) steps as usual. In contrast, if an unmasked exception is pending, i.e., $\mathsf{unmasked}_{[\,]}(M, \xi_s) = (\xi, \xi_s')$, rule ($\textsc{Bind-Raise}$) raises it, i.e., $\mathit{raise}\ \xi$, and the thread steps with buffer ξ_s' where exception ξ has been removed.

Once raised, exceptions propagate *unconditionally* via rule ($\textsc{Bind}_\xi$), i.e., no further exceptions are raised until the current one is handled.

B. Concurrency and Synchronization Variables

The modifications needed for supporting asynchronous exceptions in the concurrent semantics are minimal. Figure 14 extends the thread state th with the list of pending exceptions ξ_s and the initial masking context M. When a thread forks, the child thread *inherits* the masking context of the parent thread and runs with an initially empty list of exceptions. New rule ($\textsc{Throw}$) processes event $\mathbf{throw}_{\ell'}(\xi, n')$ by delivering exception ξ to the thread (t, ξ_s', M') identified by (ℓ', n'). Since exceptions are processed in the same order as they are delivered, the rule inserts ξ at the end of the buffer ξ_s', i.e., $\xi_s' +\!\!+ [\xi]$.

Next, we introduce new rules that capture precisely the interaction between synchronization variables and asynchronous exceptions. As we explained before, Concurrent Haskell by design allows specific *blocking* operations to be *interrupted* by asynchronous exceptions, even if they are masked [27]. Therefore, our semantics resumes threads stuck on synchronization variables if *any* exception is pending. The rules in Figure 13 formalize this requirement for primitive $\mathit{takeMVar}$, the rules for $\mathit{putMVar}$ are symmetric. Rule ($\textsc{Take}_1$) covers the case where no unmasked exception is pending, i.e., $\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset$, and the thread can step because the variable is full, i.e., $(\ell, n) \mapsto (\!| \, t \, |\!) \in \Sigma$, and thus the rule returns its content t and empties the variable, i.e., $\Sigma[(\ell, n) \mapsto \otimes]$. On the other hand, in rule ($\textsc{Take-Raise-Unmasked}$), an unmasked exception is pending, i.e., $\mathsf{unmasked}_{[\,]}(M, \xi_s) = (\xi, \xi_s')$, thus, regardless of the variable being full or empty, i.e., $(\ell, n) \mapsto c \in \Sigma$, the rule aborts the computation and raises the exception ξ without modifying the store. Lastly, in rule ($\textsc{Take-Raise-Masked}$), the variable is *empty*, i.e., $(\ell, n) \mapsto \otimes \in \Sigma$, and the thread should block and get stuck. However, an exception ξ is pending in the buffer ξ_s, i.e., $\xi_s = \xi : \xi_s'$, therefore—*regardless* of the masking context M—the thread is resumed by raising exception ξ. In the rule, the condition $\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset$ reveals that the pending exception that gets raised is *masked* and ensures that the semantics is *deterministic*. Without this premise, a thread with both unmasked and masked exceptions pending in its buffer could either step via rule ($\textsc{Take-Raise-Unmasked}$) and raise the first unmasked exception, or via rule ($\textsc{Take-Raise-Masked}$) and raise the first masked exception. The condition above removes the non-determinism: if both masked and unmasked exceptions are pending, the first *unmasked* exception is raised via rule ($\textsc{Take-Raise-Unmasked}$).

C. Design Choices and Security

In this part we motivate some of the design choices that are key to the security guarantees of $\mathbf{MAC}_{\mathit{async}}$ and that, we believe, can help programmers to write code that is more robust to asynchronous exceptions.

a) API of $\mathit{throwTo}$: The type of $\mathit{throwTo}$ (Fig. 9) restricts how threads are permitted to communicate asynchronously with each other to enforce security. Imagine an unrestricted version of $\mathit{throwTo}$ called $\mathit{throwTo}^{\mathbf{leaky}}$, which—in clear violation of the *no write-down* security rule—allows secret threads to send exceptions to public threads. If $\mathbf{MAC}_{\mathit{async}}$ exposed this leaky primitive, then an attacker could exploit it to leak secret data to a public thread through classic *implicit flows* attacks:

```
do tid_L ← fork_L (do catch loop [(ξ, print_L 1)])
   _     ← fork_H (do s ← unlabel secret
                      if s then throwTo^leaky tid_L ξ
                           else return ())
```

The code above forks two threads, a public thread that waits for an asynchronous exception in a loop, and a secret thread that branches on secret data and sends an exception to the public thread in one branch. Since the secret thread sends an exception to the public thread only when the secret is true, the attacker can easily learn its value by simply monitoring the public output of the public thread, which prints 1 only when an exception is raised. $\mathbf{MAC}_{\mathit{async}}$ rejects such attacks by statically enforcing the no write-down rule in the API of $\mathit{throwTo}$, which would make the code above ill-typed.

b) Asynchronous $\mathit{throwTo}$: In $\mathbf{MAC}_{\mathit{async}}$, primitive $\mathit{throwTo}$ is itself an asynchronous operation. As rule ($\textsc{ThrowTo}_2$) in Figure 10 shows, primitive $\mathit{throwTo}$ always returns immediately, without waiting for the receiver thread to raise the exception. This design choice follows a previous line of work on asynchronous exceptions for Haskell [27], where the authors argue that the asynchronous semantics is easier to implement. Maybe surprisingly, the current implementation of Concurrent Haskell with asynchronous exception

Thread: $th ::= (t, \xi_s, M)$

(THROW)

$$\frac{\begin{array}{c}\phi = \mathsf{nextId}(\Theta_1) \\ \Sigma, t_1, \xi_s \xrightarrow{\mathbf{throw}_{\ell'}(\xi, n')}_{(\phi, M)} \Sigma, t_2, \xi_s \qquad \Theta_2 = \Theta_1[(\ell, n) \mapsto (t_2, \xi_s, M)] \qquad (\ell', n') \mapsto (t, \xi'_s, M') \in \Theta_2\end{array}}{\ell, n \vdash \langle \Sigma, \Theta_1[(\ell, n) \mapsto (t_1, \xi_s, M)] \rangle \hookrightarrow \langle \Sigma, \Theta_2[(\ell', n') \mapsto (t, \xi'_s +\!\!+ [\xi], M')] \rangle}$$

Fig. 14: Extended concurrent semantics for asynchronous exceptions.

in the GHC runtime provides only a *synchronous* version of $\mathit{throwTo}$.[8] Would **MAC**$_{\mathit{async}}$be secure with a synchronous primitive $\mathit{throwTo}^{\mathbf{sync}}$? No, unfortunately the possibility of two threads synchronizing by throwing exceptions opens a new covert channel. Consider the following example, where $\mathit{throwTo}^{\mathbf{sync}}$ has synchronous semantics, i.e., $\mathit{throwTo}^{\mathbf{sync}}$ *blocks* the sender thread until the exception is raised in the receiver thread.

```
do tid_H ← fork_H (do s ← unlabel secret
                      if s then mask ξ loop else loop)
   no_ops
   throwTo^sync tid_H ξ
   print_L 0
```

In the code above, the main public thread forks a secret thread, which branches on secret data and in one branch enters the masked block $\mathit{mask}\ \xi\ \mathit{loop}$. After waiting for a sufficient amount of time through $\mathit{no\_ops}$, the public thread sends exception ξ *synchronously* to the secret thread. If the secret thread is looping in the masked block, the exception ξ will never be raised, causing the public thread to block forever on $\mathit{throwTo}^{\mathbf{sync}}$ and thus suppressing the final public output $\mathit{print}_{L}\ 0$. Then, the attacker can learn the value of the secret by simply observing (the lack of) the output 0 on the public channel.

As discussed in Section II for MVar, synchronous communication primitives perform both read *and* write side-effects, therefore $\mathit{throwTo}^{\mathbf{sync}}$ cannot operate securely between threads at different security levels. Even though Concurrent Haskell provides only the equivalent of $\mathit{throwTo}^{\mathbf{sync}}$, we can still derive a secure asynchronous implementation for $\mathit{throwTo}$ by internally forking an isolated thread that calls the unsafe $\mathit{throwTo}^{\mathbf{sync}}$, i.e., we define $\mathit{throwTo}\ \ t\ \ \xi$ as $\mathit{fork}\ (\mathit{throwTo}^{\mathbf{sync}}\ t\ \xi \gg \mathit{return}\ ())$.

c) Reliable exception delivery: In **MAC**$_{\mathit{async}}$, threads store the received exceptions in the buffer where they remain until raised. Importantly, the exceptions are raised following the order in which they have been delivered, thus enabling threads to react to signals in the same order as they arise. Even though multiple exceptions can be pending in the buffer, our semantics ensures that new exceptions are not raised while the thread is in an exceptional state. This choice eliminates, by design, the risk of multiple simultaneous exceptions disrupting critical code in unpredictable ways. Once handled via the matching exception handler, the code resumes normal execution and any other pending exception may be raised. This ensures that all remaining exceptions, if not masked, will eventually be raised.

D. Relation to MAC

MAC$_{\mathit{async}}$ extends **MAC** with asynchronous exceptions [23]. **MAC** features exception-handling primitives and classic exceptions, but these operate within individual threads and are intended to signal and recover from exceptional conditions arising only *internally*, due to the current state of the computation. Asynchronous exceptions are more expressive than regular exceptions. In addition to signaling (external) exceptional conditions, they enable a flexible signal-based communication mechanism. In **MAC**, threads can communicate with each other only synchronously and *indirectly*, through synchronization variables. Though possible, this communication mechanism is too cumbersome to use as it would require programmers to establish an appropriate communication protocol and change their code heavily, for example to ensure that all threads that need communicating share the same synchronization variable. Even worse, communication in this style is limited between threads at the same security level. In contrast, threads in **MAC**$_{\mathit{async}}$ can communicate *directly*, by sending exceptions to the identifier of the intended receiver thread, and also to threads at a different, more sensitive security level. **MAC**$_{\mathit{async}}$ leverages the mechanization of **MAC** in its security proofs. Modeling the semantics of asynchronous exceptions presented in this paper required substantial changes to the existing artifact. These changes include extending the syntax and semantics of the previous model with our new primitives, as well as carefully adapting the existing semantics rules to capture the semantics of *interruptible exceptions*.

VI. SECURITY GUARANTEES

This section shows that **MAC**$_{\mathit{async}}$ satisfies *progress-sensitive* non-interference (PSNI). We begin by describing our proof technique based on *term erasure*. Then, we present two lemmas that are key to the progress-sensitive guarantees of the calculus and sketch the non-interference proof. We refer the interested reader to the Agda mechanization for detailed proofs.

A. Term Erasure

Term erasure is a widely used technique to prove non-interference properties of IFC languages (e.g. [23, 26, 30, 34, 37, 38, 39]). The technique takes its name from the *erasure function*, which removes secret data *syntactically* from program terms. To this end, the erasure function, written $\varepsilon_{\ell_A}(t)$,

[8] https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Exception.html#v:throwTo

$$\begin{tikzcd}
t \arrow[r, squiggly] \arrow[d, "\varepsilon_{\ell_A}"'] & t' \arrow[d, "\varepsilon_{\ell_A}"] \\
\varepsilon_{\ell_A}(t) \arrow[r, squiggly] & \varepsilon_{\ell_A}(t')
\end{tikzcd}$$

Fig. 15: Single-step simulation.

rewrites the sub-terms of t above the attacker's security level ℓ_A to special term $\bullet$, which only reduces to itself. Once this function is defined, the technique relies on establishing a core property, a *simulation* between the execution of terms (and later configurations as well) and their erased counterpart. The simulation diagram in Figure 15 illustrates this property for pure terms. The diagram shows that erasing the confidential parts of term t and then reducing the erased term $\varepsilon_{\ell_A}(t)$ along the orange path leads to the same term $\varepsilon_{\ell_A}(t')$ obtained along the cyan path by first stepping from term t to term t' and then applying erasure, i.e., the diagram *commutes*. Intuitively, if term t leaked while stepping to t', then some data above security level ℓ_A would remain in the erased term $\varepsilon_{\ell_A}(t')$, but it would be erased along the other path, in which t is first erased and then reduced, and thus the diagram would not commute.

B. Erasure Function

We define the erasure function for terms in Figure 16. Since the sensitivity of many terms is determined by their type, the definition of the erasure function is type driven, i.e., we write $\varepsilon_{\ell_A}(t :: \tau)$ for the erasure of term t of type τ. (We omit the type of the term when it is irrelevant). Ground values are unaffected by the erasure function, e.g., $\varepsilon_{\ell_A}(()) = ()$, and most terms are erased homomorphically, e.g., $\varepsilon_{\ell_A}(t_1\ \ t_2) = \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2)$. The content of *secret* labeled values is removed, i.e., $\varepsilon_{\ell_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ \ell\ t) = \mathit{Labeled}\ \bullet$ if $\ell \not\sqsubseteq \ell_A$, or erased homomorphically otherwise, i.e., $\varepsilon_{\ell_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ \ell\ t) = \mathit{Labeled}\ \varepsilon_{\ell_A}(t : \tau)$ if $\ell \sqsubseteq \ell_A$. Notice that terms of type $\mathit{MAC}\ \ell\ \tau$ (e.g., mask, unmask) are also erased homomorphically, despite the fact that the computation may be sensitive, i.e., even if $\ell \not\sqsubseteq \ell_A$. (The special erasure for primitive $\mathit{throwTo}$ is explained below). Should not erasure rewrite these constructs to $\bullet$? Intuitively, these terms represent a *description* of a sensitive computation, which cannot leak data until it is inserted in a sequential configuration and executed. Since these terms can only execute when fetched from a thread pool, it is then sufficient to erase thread pools appropriately.

We define the erasure function for configurations, stores, thread pools, and thread states in Figure 17. Configurations are erased component-wise, i.e., $\varepsilon_{\ell_A}(\langle\Sigma, \Theta\rangle) = \langle\varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Theta)\rangle$. Thread pools Θ containing secret threads are entirely removed by the erasure function, i.e., $\varepsilon_{\ell_A}(\Theta)(\ell) = \bullet$ if $\ell \not\sqsubseteq \ell_A$, while those containing thread pools are erased homomorphically, i.e., $\varepsilon_{\ell_A}(\Theta)(\ell) = \varepsilon_{\ell_A}(\Theta(\ell))$ if $\ell \sqsubseteq \ell_A$, where $\varepsilon_{\ell_A}([\,]) = [\,]$ and $\varepsilon_{\ell_A}(th, T_s) = (\varepsilon_{\ell_A}(th), \varepsilon_{\ell_A}(T_s))$. (The erasure function for memory stores and segments is similar). When some secret thread gets scheduled from an erased thread pool $\bullet$, a *dummy* thread $(\bullet, [\,], \lambda\_.\mathit{False})$ runs instead and simply loops. However, rewriting secret thread pools to $\bullet$ can disrupt operations involving thread identifiers. For example, an erased public thread using primitive $\mathit{throwTo}$ to communicate with a secret thread gets *stuck*, since rule (THROW) would try to deliver an exception into thread pool $\bullet$. To recover from this situation, we apply the *two-step erasure* technique [30]. This technique rewrites problematic terms, e.g., $\mathit{throwTo}$, to special, $\bullet$-annotated erased terms added to the calculus, i.e., $\mathit{throwTo}_\bullet$. The semantics of these *new* terms is then defined precisely to re-establish the core simulation property fundamental for security (Fig. 15). For example, term $\mathit{throwTo}_\bullet\ t\ \xi$ reduces just like $\mathit{thorwTo}$ in rules (THROWTO$_1$) and (THROWTO$_2$), thus respecting the simulation property of the sequential semantics. However, instead of generating a regular event $\mathbf{throw}_{\ell_H}(\xi, n)$, which would get the concurrent configuration stuck in rule (THROW), it generates a new event $\mathbf{throw}_{\bullet\ell_H}(\xi)$. Similarly, this event is handled by a new rule of the concurrent semantics, which simply drops the exception (no thread labeled $\ell_H \not\sqsubseteq \ell_A$ can receive it), thus completing the simulation diagram of the concurrent step (THROW).

$$\varepsilon_{\ell_A}(()) = ()$$
$$\varepsilon_{\ell_A}(t_1\ t_2) = \varepsilon_{\ell_A}(t_1)\ \ \varepsilon_{\ell_A}(t_2)$$
$$\varepsilon_{\ell_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ \ell\ t) = \begin{cases} \mathit{Labeled}\ \varepsilon_{\ell_A}(t) & \text{if } \ell_{\mathrm{H}} \sqsubseteq \ell_A \\ \mathit{Labeled}\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{\ell_A}(\mathit{mask}\ \xi\ t) = \mathit{mask}\ \xi\ \ \varepsilon_{\ell_A}(t)$$
$$\varepsilon_{\ell_A}(\mathit{unmask}\ \xi\ t) = \mathit{unmask}\ \xi\ \ \varepsilon_{\ell_A}(t)$$
$$\varepsilon_{\ell_A}(\mathit{throwTo}\ (t :: \mathit{TId}\ \ell_{\mathrm{H}})\ \xi :: \mathit{MAC}\ \ell_{\mathrm{L}}\ ()) = \begin{cases} \mathit{throwTo}\ \varepsilon_{\ell_A}(t)\ \ \xi & \text{if } \ell_{\mathrm{H}} \sqsubseteq \ell_A \\ \mathit{throwTo}_\bullet\ \varepsilon_{\ell_A}(t)\ \ \xi & \text{otherwise} \end{cases}$$

Fig. 16: Erasure of terms (excerpts).

$$\varepsilon_{\ell_A}(\langle\Sigma, \Theta\rangle) = \langle\varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Theta)\rangle$$
$$\varepsilon_{\ell_A}(\Sigma)(\ell) = \begin{cases} \varepsilon_{\ell_A}(S) & \text{if } \ell \sqsubseteq \ell_A \text{ and } S = \Sigma(\ell) \\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{\ell_A}(\Theta)(\ell) = \begin{cases} \varepsilon_{\ell_A}(T_s) & \text{if } \ell \sqsubseteq \ell_A \text{ and } T_s = \Theta(\ell) \\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{\ell_A}((t, \xi_s, M)) = (\varepsilon_{\ell_A}(t), \xi_s, M)$$

Fig. 17: Erasure of configurations (excerpts)

C. Progress-Sensitive Non-Interference

The proof of progress-sensitive non-interference builds on two key properties of the concurrent relation: *deterministic reduction* and *erased simulation*.

Lemma 1 (Deterministic Reduction)**.** *If $c_1 \hookrightarrow c_2$ and $c_1 \hookrightarrow c_3$, then $c_2 \equiv c_3$.*

The symbol $\equiv$ above denotes syntactic equality up to α-renaming, in our mechanized proofs we use De Bruijn indexes and syntactic equality. Determinism of the concurrent semantics is important for security, because it eliminates scheduler refinement attacks [13].

The second lemma that we prove relates the reduction step of a thread in the concurrent semantics with the corresponding erased thread. If the security level ℓ of the thread is below the level of the attacker, i.e. $\ell \sqsubseteq \ell_A$, then we construct a simulation diagram similar to that of Figure 15, but for concurrent steps. Instead, if the security level of the thread is *not* observable by the attacker, i.e., $\ell \not\sqsubseteq \ell_A$, then the configurations before and after the step are *indistinguishable* to the attacker. This indistinguishability relation is called ℓ_A-equivalence, written $c_1 \approx_{\ell_A} c_2$, and defined as the kernel of the erasure function (Fig. 17), i.e., $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$.

Lemma 2 (Erased Simulation)**.** *Given a concurrent reduction step $\ell, n \ \vdash c_1 \hookrightarrow c_1'$ then*

- *$\ell, n \ \vdash \varepsilon_{\ell_A}(c_1) \hookrightarrow \varepsilon_{\ell_A}(c_1')$, if $\ell \sqsubseteq \ell_A$, or*
- *$c_1 \approx_{\ell_A} c_1'$, if $\ell \not\sqsubseteq \ell_A$*

Using Lemmas 1 and 2, we prove progress-sensitive non-interference (PSNI), where symbol $\hookrightarrow^*$ denotes the transitive reflexive closure of $\hookrightarrow$ as usual.

Theorem 1 (Progress-Sensitive Non-Interference)**.** *Given two well-typed concurrent configurations c_1 and c_2, such that $c_1 \approx_{\ell_A} c_2$, and a reduction step $\ell, n \ \vdash c_1 \hookrightarrow c_1'$, then there exists a configuration c_2' such that $c_1' \approx_{\ell_A} c_2'$ and $c_2 \hookrightarrow^* c_2'$.*

Proof. By cases on $\ell \sqsubseteq \ell_A$.

If $\ell \sqsubseteq \ell_A$ then in the configuration c_2 there is an ℓ_A-equivalent thread identified by (ℓ, n). Before that thread runs, however, there can be a *finite* number of high threads in c_2 scheduled before (ℓ, n). After the high threads run, i.e. $c_2 \hookrightarrow^* c_2''$, for some configuration c_2'', the low thread is scheduled again, i.e. $\ell, n \ \vdash c_2'' \hookrightarrow c_2'$, for some other configuration c_2'. From Lemma 2 (*erased simulation*) applied to the first set of steps, we obtain $c_2 \approx_{\ell_A} c_2''$ (all these steps involve threads above the attacker's level) and then $c_1 \approx_{\ell_A} c_2''$ follows by transitivity of the ℓ_A-equivalence relation. Then, we apply Lemma 2 again and conclude that $\ell, n \ \vdash \varepsilon_{\ell_A}(c_2'') \hookrightarrow \varepsilon_{\ell_A}(c_2')$, since $\ell \sqsubseteq \ell_A$ as well as $\ell, n \ \vdash \varepsilon_{\ell_A}(c_1) \hookrightarrow \varepsilon_{\ell_A}(c_1')$. By definition of ℓ_A-equivalence, we derive $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2'')$ from $c_1 \approx_{\ell_A} c_2''$ and from Lemma 1 (*determinancy*) we conclude that $\varepsilon_{\ell_A}(c_1') \equiv \varepsilon_{\ell_A}(c_2')$, i.e., $c_1' \approx_{\ell_A} c_2'$.

If $\ell \not\sqsubseteq \ell_A$, then we apply Lemma 2 and obtain $c_1 \approx_{\ell_A} c_1'$, thus $c_1' \approx_{\ell_A} c_2'$ for $c_2' = c_2$ by reflexivity and transitivity of ℓ_A-equivalence. □

VII. Related Work

Asynchronous Exceptions Mechanisms. Asynchronous exceptions and signals allow developers to implement key functionalities of real-world systems (e.g., speculative computation, timeouts, user interrupts, and enforcing resources bounds) robustly [27, 40]. Surprisingly, support for asynchronous exceptions in concurrent programming languages differ considerably. For example, Java has deprecated fully asynchronous methods to stop, suspend, and resume threads because they can too easily break programs invariants without hope of recovery [21]. Similarly, the interaction between synchronous exceptions and signals makes it hard to write robust signal handlers in Python [20]. Lacking robust asynchronous primitives, several programming languages and operating systems (e.g., Java, Modula-3, and POSIX-compliant OS'es) rely on *semi-asynchronous* communication as a workaround. With this approach, a thread sends a signal to another by setting special flags that must be polled periodically by the receiver. Even though programming in this model is less convenient, we believe that the principles proposed in this paper could be adapted for semi-asynchronous communication. The Standard ML of New Jersey (SML/NJ) features asynchronous signaling mechanisms based on first-class continuations [28]. When a thread receives a signal, control is passed to the corresponding handler together with the interrupted state of the thread as a continuation. Then, the handler may decide to resume the execution of the interrupted thread or pass control to another thread. Erlang implements a special kind of asynchronous signaling. Threads can monitor each other through *bidirectional links*, which propagate the exit code of a thread to the other [41]. Multicore OCaml support asynchronous exceptions through algebraic effects and effects handlers [42]. Syme et al. [43] extend F# with an asynchronous modality that changes the semantics of control-flow operators to use continuations, thus sparing programmers from writing asynchronous code in continuation-passing style. Bierman et al. [44] port this approach to C# and additionally formalize it with a direct operational semantics and prove type-safety. Inoue et al. [45] provide interruptible executions in Scala for context-aware (reactive) programming via an embedded domain specific language based on workflows. Concurrent Haskell supports asynchronous exceptions with scoped (un)masking combinators [27] and **MAC**$_{async}$ relies on them to provide secure API to untrusted code.

Semantics of Asynchronous Exceptions. Peyton Jones et al. [46] present a semantics framework for reasoning about the correctness of compiler optimizations in the presence of (imprecise) exceptions for Haskell. Their framework can capture asynchronous exceptions as well, but it is based on denotational semantics and thus not suitable for reasoning about covert channels. Marlow et al. [27] were the first to develop an operational semantics for asynchronous exceptions, which inspired ours and which we have extended to model fine-grained exception handlers. Their semantics is based on evaluation contexts [47], while ours is small-step to leverage the existing formalization and mechanization of **MAC** [15, 23, 30]. Hutton and Wright [48] study an operational semantics for interrupts in the context of a basic terminating language without concurrency and I/O. Their goal is exploratory: they want to formally justify the source level semantics with respect to its compilation to a low-level language. Harrison et al. [49] identify asynchronous exceptions as a computational effect and formalize them in a modular monadic model.

Covert Channels and Countermeasures. Several runtime system features create subtle covert channels that weaken and sometimes completely break the security guarantees of IFC languages. When memory is shared between computations at different security levels, *garbage collection* cycles leak information via timing, even across network connections [10]. To close this channel, memory should be partitioned by security level and each memory partition should be managed by an independent timing-sensitive garbage collector (see the garbage collector implemented in Zee for an example [12]). *Lazy evaluation* introduces a software level cache in the runtime system which creates an internal timing channel in concurrent Haskell IFC libraries [9]. To close this channel, Vassena et al. [15] design a runtime system primitive that *restricts sharing* between threads at different security levels. The same primitive can close the lazy covert channel in $\textbf{MAC}_{\mathit{async}}$. General purpose runtime system automatically balance computing resources (CPU time, memory and cores) between running threads to achieve fairness, but, by doing so on multi-core systems, they also internalize many external timing covert channels [11]. LIO_{PAR} is a runtime system design that recovers security in multi-core systems by making resource management hierarchical and explicit at the programming language level. Even though in LIO_{PAR} parent threads send asynchronous signals to kill children and reclaim computing resources, LIO_{PAR} does not support fine-grained exception handlers and masking primitives. Language-based predictive mitigation is a general technique to bound the leakage of timing-channels (e.g., arising due to hardware caches) in programs [50]. Jérémy and Aslan [51] optimize this technique for a sequential programming language with asynchronous I/O, but their approach does not consider concurrency and asynchronous exceptions. Interruptible enclaves have been the target of several attacks interrupt-based attacks [52, 53, 54] and Busi et al. [55] propose *full abstraction* [56] as the desirable security criterion for extending processor with interruptible enclaves securely. Our security criterion (progress-sensitive non-interference) is simpler to prove and aligns with the expected security guarantees of $\textbf{MAC}_{\mathit{async}}$. Intuitively, Busi et al. [55] prove a more complex criterion because it ensures that the extended processor has no more vulnerabilities than the original, but that does not imply that neither processor satisfies some specific security property.

Secure Runtime Systems and Abstractions. Systems that by design run untrusted programs (e.g., mobile code and plug-ins) must place adequate security mechanisms to impede buggy or malicious code from exhausting all available computing resources. KaffeOS is an extension of the Java runtime system that isolates *processes* and manage their computing resources (memory and CPU time) to prevent abuse [57]. When a process exceeds its resource budget, KaffeOS kills it and reclaims its resources without affecting the integrity of the system. Cinder is an operating system for mobile devices that provides *reserves* and *taps* abstractions for storing and distributing energy [58]. Using these abstractions, applications can delegate and subdivide their energy quota while maintaining energy isolation. Yang and Mazières [59] extend GHC runtime systems with *resource containers*, an abstraction that enforce dynamic space limits according to an allocator-pays semantics. None of these systems enforce information flow policies except for Cinder, but we believe that secure API for asynchronous exceptions like those of $\textbf{MAC}_{\mathit{async}}$ could represent a basic building block to enforce them reliably.

Zee is an IFC language for implementing secure (timing-sensitive) runtime systems [12]. The lack of asynchronous exceptions in Zee complicates the implementation of certain runtime system components, but we believe that Zee could support them by applying the insights from this work. An interesting line of work aims at exposing safe high-level API to allow users to reprogram features of the runtime systems, e.g., concurrency primitives [17], multi-core schedulers [18], and kill-safe abstractions [19]. We believe that the primitives designed to remove covert channels in GHC and other runtime systems discussed above could be implemented following this approach.

VIII. Conclusions and Future Work

This work presents the first IFC language that support asynchronous exceptions securely. Embedded in Haskell, the IFC library $\textbf{MAC}_{\mathit{async}}$ provides primitives for fine-grained masking and unmasking of asynchronous exceptions, which enable useful programming patterns, that we showcased with two examples. We have formalized $\textbf{MAC}_{\mathit{async}}$ in 3,000 lines of Agda and proved progress-sensitive non-interference.

As future work, we plan to use $\textbf{MAC}_{\mathit{async}}$ to reason about the delivery of OS signals to threads. Specially, we will explore OS signals dedicated to alert about exhaustion of resources that cannot be easily partitioned (e.g., the battery in an IoT board). This scenario will demand the OS—which can be thought as just another thread—to send signals from higher to lower levels in the security lattice, thus opening up an information leakage channel which, we believe, needs to be mitigated.

Another direction for future work consists on using $\textbf{MAC}_{\mathit{async}}$ to build realistic systems. For instance, we expect $\textbf{MAC}_{\mathit{async}}$ to be able to provide an IFC-aware interface for GHC to control CPU usage by leveraging on previous work [17, 18]. Moreover, building realistic systems often involves mutually distrusts principals, where we expect *privileges* [60, 61] to restrict untrusted code from abusing our selective mask mechanism.

Acknowledgments

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023) as well as the Swedish research agency Vetenskapsrådet. This work was partially supported by the German Federal Ministry of Education and

Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity.

References

[1] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[2] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.

[3] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating system," in *ACM Symposium on Operating Systems Principles*, ser. SOSP. ACM, 2005.

[4] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining JavaScript with COWL," in *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2014.

[5] A. Yip, N. Narula, M. Krohn, and R. Morris, "Privacy-preserving browser-side scripting with BFlow," in *EuroSys*, 2009.

[6] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in JavaScript and its APIs," in *ACM Symposium on Applied Computing*. ACM, 2014.

[7] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java Information Flow," 2001, http://www.cs.cornell.edu/jif.

[8] A. Russo, K. Claessen, and J. Hughes, "A library for light-weight information-flow security in Haskell," in *ACM SIGPLAN symposium on Haskell*. ACM, 2008.

[9] P. Buiras and A. Russo, "Lazy programs leak secrets," in *Nordic Conference in Secure IT Systems*. Springer-Verlag, 2013.

[10] M. V. Pedersen and A. Askarov, "From trash to treasure: Timing-sensitive garbage collection," in *2017 IEEE Symposium on Security and Privacy*, 2017.

[11] M. Vassena, G. Soeller, P. Amidon, M. Chan, J. Renner, and D. Stefan, "Foundations for parallel information flow control runtime systems," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds. Cham: Springer International Publishing, 2019, pp. 1–28.

[12] M. Vorreiter Pedersen and A. Askarov, "Static enforcement of security in runtime systems," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, June 2019, pp. 335–33 515.

[13] A. Russo and A. Sabelfeld, "Securing Interaction between Threads and the Scheduler," in *IEEE Computer Sec. Foundations Workshop*, 2006, pp. 177–189.

[14] ——, "Securing timeout instructions in web applications," in *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 2009, pp. 92–106.

[15] M. Vassena, J. Breitner, and A. Russo, "Securing concurrent lazy programs against information leakage," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, Aug 2017, pp. 37–52.

[16] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018.

[17] P. Li, S. Marlow, S. Peyton Jones, and A. Tolmach, "Lightweight concurrency primitives for GHC," in *Proc. of the ACM SIGPLAN Workshop on Haskell Workshop*, ser. Haskell '07. ACM, 2007.

[18] K. C. Sivaramakrishnan, T. Harris, S. Marlow, and S. Petyon Jones, "Composable scheduler activations for Haskell," *Journal of Functional Programming*, vol. 26, 2016.

[19] M. Flatt and R. B. Findler, "Kill-safe synchronization abstractions," *SIGPLAN Not.*, vol. 39, no. 6, p. 47–58, Jun. 2004.

[20] F. N. Stephen and M. P. Mark, "Safe Asynchronous Exceptions For Python," Williams College, Tech. Rep., 12 2002.

[21] Oracle, "Java Thread Primitive Deprecation," https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html, accessed on 05.02.2020.

[22] A. Russo, "Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell," in *ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP. ACM, 2015.

[23] M. Vassena, A. Russo, P. Buiras, and L. Waye, "MAC a verified static information-flow control library," *Journal of Logical and Algebraic Methods in Programming*, vol. 95, pp. 148–180, 2018.

[24] D. Hedin and A. Sabelfeld, "A Perspective on Information-Flow Control," in *2011 Marktoberdorf Summer School*. IOS Press, 2011.

[25] Agda development team, "Agda 2.6.0.1 documentation," https://agda.readthedocs.io/en/v2.6.0.1/, 2019.

[26] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *ACM SIGPLAN Haskell symposium*, 2011.

[27] S. Marlow, S. P. Jones, A. Moran, and J. Reppy, "Asynchronous exceptions in Haskell," in *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, 2001, pp. 274–285.

[28] J. H. Reppy, "Asynchronous Signals in Standard ML," Tech. Rep., 1990.

[29] R. P. Gabriel and J. McCarthy, "Queue-based multi-processing LISP," in *Proc. of ACM Symposium on LISP and Functional Programming*, ser. LFP '84. ACM, 1984, p. 25–44.

[30] M. Vassena and A. Russo, "On formalizing information-flow control libraries," in *Proc. of ACM Workshop on Programming Languages and Analysis for Security*, ser.

PLAS '16. ACM, 2016.

[31] M. Vassena, P. Buiras, L. Waye, and A. Russo, "Flexible manipulation of labeled values for information-flow control libraries," in *Proceedings of the 12th European Symposium On Research In Computer Security*. Springer, Sep. 2016.

[32] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[33] D. E. Bell and L. La Padula, "Secure computer system: Unified exposition and multics interpretation," MITRE Corporation, Bedford, MA, Tech. Rep. MTR-2997, Rev. 1, 1976.

[34] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres, "Addressing covert termination and timing channels in concurrent information flow systems," in *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2012.

[35] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *ACM symposium on Principles of Programming Languages*, 1998.

[36] S. Peyton Jones, A. Gordon, and S. Finne, "Concurrent Haskell," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.

[37] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical Computer Science*, vol. 411, no. 19, pp. 1974–1994, 2010.

[38] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo, *IFC Inside: Retrofitting Languages with Dynamic Information Flow Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015.

[39] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.

[40] M. Simon, "Asynchronous Exceptions in Practice," https://simonmar.github.io/posts/2017-01-24-asynchronous-exceptions.html, 2017.

[41] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[42] S. Dolan, S. Eliopoulos, D. Hillerström, A. Madhavapeddy, K. C. Sivaramakrishnan, and L. White, "Concurrent system programming with effect handlers," in *Trends in Functional Programming*, M. Wang and S. Owens, Eds. Cham: Springer International Publishing, 2018, pp. 98–117.

[43] D. Syme, T. Petricek, and D. Lomov, "The F# Asynchronous Programming Model," in *Proceedings of Practical Aspects of Declarative Languages*, ser. PADL 2011, 2011.

[44] G. M. Bierman, C. V. Russo, G. Mainland, E. Meijer, and M. Torgersen, "Pause 'n' play: Formalizing asynchronous c#," in *ECOOP 2012 - Object-Oriented Programming Conference*, 2012, pp. 233–257.

[45] H. Inoue, T. Aotani, and A. Igarashi, "ContextWorkflow: A Monadic DSL for Compensable and Interruptible Executions," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.

[46] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow, "A semantics for imprecise exceptions," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '99. ACM, 1999.

[47] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theor. Comput. Sci.*, vol. 103, no. 2, p. 235–271, Sep. 1992.

[48] G. Hutton and J. Wright, "What is the meaning of these constant interruptions?" *Journal of Functional Programming*, vol. 17, no. 6, p. 777–792, 2007.

[49] W. L. Harrison, G. Allwein, A. Gill, and A. Procter, "Asynchronous exceptions as an effect," in *Mathematics of Program Construction*, P. Audebaud and C. Paulin-Mohring, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 153–176.

[50] D. Zhang, A. Askarov, and A. C. Myers, "Language-based Control and Mitigation of Timing Channels," in *ACM Conference on Programming Language Design and Implementation*. ACM, 2012.

[51] T. Jérémy and A. Aslan, "Language-based predictive mitigation for systems with asynchronous I/O," Tech. Rep., 2017.

[52] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-step: A practical attack framework for precise enclave execution control," in *Proc. of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. ACM, 2017.

[53] W. He, W. Zhang, S. Das, and Y. Liu, "Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, Oct 2018, pp. 108–114.

[54] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018.

[55] M. Busi, J. Noorman, J. V. Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens, "Provably secure isolation for interruptible enclaved execution on small microprocessors," in *IEEE Computer Security Foundations Symposium*, ser. CSF. IEEE Computer Society, 2020.

[56] M. Abadi, *Protection in Programming-Language Translations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 19–34.

[57] G. Back and W. C. Hsieh, "The KaffeOS Java runtime system," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 4, p. 583–630, 2005.

[58] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich, "Energy management in mobile devices with the cinder operating system," in *Proc. of the Sixth Conference on Computer Systems*, ser. EuroSys '11.

ACM, 2011, p. 139–152.

[59] E. Z. Yang and D. Mazières, “Dynamic space limits for Haskell,” in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. ACM, 2014.

[60] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, “Disjunction category labels,” in *Nordic Conference on Information Security Technology for Applications (NORDSEC ’11)*. Springer-Verlag, 2011.

[61] L. Waye, P. Buiras, D. King, S. Chong, and A. Russo, “It’s my privilege: Controlling downgrading in DC-labels,” in *International Workshop on Security and Trust Management*, 2015.