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\textsubscript{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\textsubscript{async} satisfy progress-sensitive non-interference and mechanize our formal claims in the Agda proof assistant.

I. INTRODUCTION

Information-Flow Control \cite{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., Linux \cite{2,3}), web browsers (e.g., Firefox \cite{4,5}), and several programming languages (e.g., Java \cite{6,7,8}).

Most language-based approaches for IFC reason about constructs found in programs (e.g., variables, branches, and data structures), while often ignoring aspects of runtime systems which might create covert channels (e.g., \cite{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 \cite{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 \cite{10}, the scheduler \cite{13}, timeouts \cite{14}, lazy evaluation \cite{15} and caches \cite{16}. 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 \cite{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-averse 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 \textit{signals} as a limited form of inter process communication (IPC)\footnote{In this last case, we are abusing the term runtime to denote “the rest of the system.”}. 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 \textit{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 \cite{20} and Java \cite{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: \textit{speculative execution} (i.e., a thread can spawn a child thread and later decide that it does not need the result

\url{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 (PSN) [24] for MAC_async and provide mechanized proofs in Agda [28] 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., [28, 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.
- 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.

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 \( \mathcal{L} = \{L, H\} \), \( \subseteq \) classifies data as either public (L) or secret (H) and only prohibits sending secret inputs into public outputs, i.e., \( H \nsubseteq L \). 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 data-type and valid flows of information (the \( \subseteq \)-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 \( \text{Labeled } \ell \tau \) associates label \( \ell \) with data of type \( \tau \). For example, \( \text{pwd :: Labeled } H \text{ String} \) is a secret string and \( \text{score :: Labeled } L \text{ Int} \) is a public integer. The abstract type \( MAC \ell \tau \) represents a side-effectful computation that manipulates data labeled with \( \ell \) and whose result has type \( \tau \). MAC provides a monadic interface to help programmers write secure code. The basic primitives of the interface are return and bind (as written by the infix operator \( >>= \)).

Primitive return \( :: \tau \rightarrow MAC \ell \tau \) creates a computation that simply returns a value of type \( \tau \) without causing side-effects. Primitive \( >>= :: MAC \ell_1 \tau_1 \rightarrow (\tau_1 \rightarrow MAC \ell_2 \tau_2) \rightarrow MAC \ell_2 \tau \) chains two computations (at the same security level \( \ell \)) together, in a sequence. Specifically, program \( m >>= f \) takes the result obtained by executing \( m \) and binds it to function \( f \), which

---

Available at [https://bitbucket.org/carlostome/mac-async](https://bitbucket.org/carlostome/mac-async)
produces the rest of the computation. Our examples use do annotation, Haskell syntactic sugar for monadic computations. For instance, we write do x ← m; return (x + 1) for the program m ≫= λx → 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 Labeled data. Computations cannot write and read labeled data directly, but must use special functions label and 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 new, ℓH-labeled value as long as the decision to do so depends on less sensitive data, i.e., the computation is labeled with ℓL such that ℓL ⊑ ℓH. (To help readers, we use subscripts in metavariables ℓL and ℓH to indicate that ℓL ⊑ ℓH). Dually, function unlabel allows ℓH-labeled computations to read data from lower security levels, i.e., data labeled with ℓL such that ℓL ⊑ ℓH. In the type-signatures of these functions, the precondition ℓL ⊑ ℓH 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 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 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 ⊑ 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 [54]. 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 ℓL-labeled computation to fork a thread at a higher security level, i.e., labeled with ℓH such that ℓL ⊑ ℓH. 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 MVar ℓ τ (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 newMVar, putMVar and takeMVar. Function newMVar creates a synchronization variable initially full with the given value. (Like label and fork, function newMVar performs a write side-effect, thus its type signature has a similar security check). Functions putMVar and 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 putMVar writes a value into an empty variable and blocks the thread if the variable is full. Dually, function takeMVar empties a full variable and returns its content and blocks the caller otherwise. Notice that putMVar and 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. MACasync 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 MACasync, which extends MAC with three new primitives 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 synchronization using asynchronous exceptions. First, we declare \( \text{Kill} :: \text{Exception} \) as a new exception, and define \( \text{kill} \ t \), a function that sends exception \( \text{Kill} \) asynchronously to the thread identified by \( t \).

\[
\begin{align*}
1 & \text{data Exception} = \text{Kill} \mid \ldots \\
2 & \text{kill} \ t = \text{throwTo} \ t \text{ Kill}
\end{align*}
\]

Then, we define the combinator \( \text{speculate} \), which receives two computations \( c_1 \) and \( c_2 \) and runs speculatively.

\[
\begin{align*}
3 & \text{speculate} :: \text{MAC } \ell \ a \rightarrow \text{MAC } \ell \ a \rightarrow \text{MAC } \ell \ a \\
4 & \text{speculate} \ c_1 \ c_2 = \text{do} \\
5 & \hspace{1em} m \leftarrow \text{newEmptyMVar} \\
6 & \hspace{1em} t_1 \leftarrow \text{fork} \ (c_1\Rightarrow \text{putMVar} \ m) \\
7 & \hspace{1em} t_2 \leftarrow \text{fork} \ (c_2\Rightarrow \text{putMVar} \ m) \\
8 & \hspace{1em} r \leftarrow \text{takeMVar} \ m \\
9 & \hspace{1em} \text{kill} \ t_1; \text{kill} \ t_2 \\
10 & \hspace{1em} \text{return} \ r
\end{align*}
\]

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 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 \( \text{Input}_{\ell} \text{ secret} \) 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 \( \text{Kill} \) 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 \( \text{initTP} \) (Fig. 3) to initialize the thread pool and start the controller thread. Function \( \text{initTP} \ n \ f \) 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 \( \text{mask} \) exception \( \text{Kill} \) while processing data, thus ensuring that they always complete ongoing computations without aborting prematurely. It may seem erroneous to mask also instruction \( \text{takeMVar} \): 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 \( \text{takeMVar} \)) are interruptible, i.e., they can receive and raise asynchronous exceptions even in masked blocks \( [27] \).

Function \( \text{controller} \ ts \ m \) 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 \( \text{wait} \) (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 (\( \text{Input}_{\ell} \text{ secret} \)), 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 \( \text{full} \) at this point, then some previous input is still waiting to be processed (all worker 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 \( \text{Kill} \) and \( \text{Input}_{\ell} \) (line 18) while processing requests. For example, if exceptions were not properly masked

\[
1 \text{type Data} = \ldots
2 \text{data Exception} = \text{Kill} \mid \text{Kill} (\text{Labeled } \ell \text{ Data}) \mid \ldots
3 \text{type Size} = \text{Int}
4 \text{initTP} :: \text{Size} \rightarrow (\text{Data} \rightarrow \text{MAC } H () \rightarrow \text{MAC } L (\text{TId } H))
5 \text{initTP} \ n \ f = \text{do}
6 \hspace{1em} m \leftarrow \text{newEmptyMVar}
7 \hspace{1em} \text{ts} \leftarrow \text{forM} [1 \ldots n] (\lambda_\ell \rightarrow \text{fork} \ (\text{worker} \ f \ m))
8 \hspace{1em} \text{fork} \ (\text{controller} \ ts \ m)
9 \hspace{1em} \text{worker} :: (\text{Data} \rightarrow \text{MAC } H()) \rightarrow \text{MVar } H \text{ Data} \rightarrow \text{MAC } H()
10 \hspace{1em} \text{worker} \ f \ m = \text{do}
11 \hspace{1em} \text{mask} [\text{Kill} (\text{takeMVar} \ m \gg f)]
12 \hspace{1em} \text{worker} \ f \ m
13 \hspace{1em} \text{controller} :: (\text{TId } H) \rightarrow \text{MVar } H \text{ Data} \rightarrow \text{MAC } H()
14 \hspace{1em} \text{controller} \ ts \ m =
15 \hspace{2em} \text{let} \ \text{wait} \ = \text{newEmptyMVar} \gg \text{takeMVar} \ \text{in}
16 \hspace{2em} \text{catch} \ \text{wait}
17 \hspace{3em} [(\text{Input}_{\ell} \text{ secret},
18 \hspace{4em} \text{mask} [\text{Input}_{\ell} \text{ secret}],
19 \hspace{5em} \text{Kill}]
20 \hspace{3em} (\text{do} \ s \leftarrow \text{unlabel} \text{ secret}
21 \hspace{4em} \text{putMVar} \ m \ s
22 \hspace{5em} \text{unmask} [\text{Input}_{\ell} \text{ secret}, \text{Kill}] (\text{controller} \ ts \ m)))]
23 \hspace{2em} , (\text{Kill},
24 \hspace{3em} \text{mask} [\text{Input}_{\ell} \text{ Kill}] (\text{forM} \ ts \ \text{kill}))
\]

Fig. 3: Thread pool example.

\[\text{In MAC}_{\text{async}}, \text{primitive } \text{fork} \text{ returns the identifier of the child thread to the parent.}\]
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 un_masks 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.

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 IPC 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.

5In `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.

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, `t₁ ⇝ t₂`, which denotes that term `t₁` reduces in one step to `t₂`. 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 `t₁ ⇝ t₂` denotes a single sequential step that brings term `t₁` of type `MAC ℓ τ` to `t₂`. Rule (UNLABEL₁) reduces term `unlabel t₁` to `unlabel t₂` by evaluating the argument through a pure semantics step `t₁ ⇝ t₂`. When the argument is evaluated, rule (UNLABEL₂) 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` (∅). A value
MV ar n denotes a synchronization memory variable that refers to the n-th cell of the l-labeled memory segment in the store.

In rule (NEW), primitive newMV ar t allocates a new memory cell containing term t in the l-labeled segment of the store, at fresh address n = l(ℓ), i.e., l(ℓ) = (ℓ, n) ⊳ (t), and returns the corresponding synchronization variable MV ar t n. Term putMV ar t1 t2 writes term t2 to the empty cell pointed by the synchronization variable t1. To do that, rule (PUT1) starts evaluating the variable t1 through a pure semantics step t1 ⊳ t1′. When the variable is fully evaluated, e.g., MV ar n, rule (PUT2) takes over and writes the given term t2 in the cell identified by (ℓ, n), i.e., l(ℓ) = (ℓ, n) ⊳ (t). Notice that the term steps only if the cell in the store is initially empty, i.e., (ℓ, n) ⊳ ⊥ ∈ l. 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 takeMV ar, which follow a similar pattern.

C. Concurrency

Unlike previous concurrent incarnations of MAC, threads in MAC 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 async incorporates thread identifiers with values TId n of the new primitive type TId l, 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.

\[
\text{fork} :: t_L \\ l_H \Rightarrow MAC l_H () \Rightarrow MAC t_L (TId l_H)
\]

Figure 7 extends the sequential calculus of MAC async with concurrency primitives. To simplify our security analysis, term fork t is annotated with the security label l of thread t of type MAC l ()). 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 fork t indicates that thread t at security level l has been forked and event 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., fork t, we use the label annotation t to generate a fresh identifier n = φ(l), which is then returned in the monad wrapped in the constructor of thread identifiers, i.e., TId n.

Figure 8 introduces the top-level semantics relation that formalizes how concurrent configurations evolve. Concurrent configurations are pairs (l, Θ) consisting of the concurrent store l 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 nextId(Θ) = l.Θ(l).

A concurrent step ℓ, n ⊢ c1 → c2 indicates that configuration c1 steps to c2, 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 async carry over for a wide range of deterministic schedulers (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., Θ[(ℓ, n) → t1], which steps silently, i.e., generating event

<table>
<thead>
<tr>
<th>Store:</th>
<th>Σ ∈ Label → Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory:</td>
<td>S ::= []</td>
</tr>
<tr>
<td>Cell:</td>
<td>c ::= ⊥</td>
</tr>
<tr>
<td>Addresses:</td>
<td>n ∈ N</td>
</tr>
<tr>
<td>Types:</td>
<td>τ ::= ···</td>
</tr>
<tr>
<td>Values:</td>
<td>v ::= ···</td>
</tr>
<tr>
<td>Terms:</td>
<td>t ::= ···</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(NEW)</td>
<td>n = [Σ(ℓ)]</td>
</tr>
</tbody>
</table>

\[
Σ, newMV ar t \xrightarrow{\text{PUT1}} Σ[(ℓ, n) ⊳ (t)], return (MVar n) \]

\[
(\text{PUT2})
\]

\[
Σ, putMV ar t1 t2 \xrightarrow{\text{ΠT2}} Σ, putMV ar t1 t2 \]

\[
\text{(FORK)}
\]

\[
n = φ(l)
\]

\[
Σ, fork t \xrightarrow{\text{fork}(t) ⊢ φ} Σ, return (TId n)
\]

Fig. 6: Syntax and semantics for synchronization variables.
deliver exception $\xi$ to the thread identified by $(\ell, n)$. To model how asynchronous exceptions propagate precisely, we add new rules both to the sequential and concurrent semantics. Rule (\text{THROWTo}) evaluates the thread identifier in term \text{throwTo} $t_1 \xi$, which reduces to \text{throwTo} $t_2 \xi$ through the pure step $t_1 \leadsto t_2$. (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 $TId\ n$, rule (\text{THROWTo}) generates event \text{throw}$(\xi, n)$ and returns unit. The rule reflects the \texttt{non-blocking} behavior of \text{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 (\text{catch}) executes the computation $t_1$ in term catch $t_1\ hs$. If during the execution of $t_1$ the computation receives some exception $\xi$, and the exception propagates up to the exception handler, then the term reduces to catch (raise $\xi$) $hs$ and rules (\text{Catch}_1) and (\text{Catch}_2) determine whether the exception gets handled or not. In these rules, function first$(hs, \xi)$ searches for a handler corresponding to exception $\xi$ 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 $\xi$. If a handler for that exception is in the list, i.e., $h = \text{first}(hs, \xi)$, then (\text{Catch}_1) passes control to it. If no handler matches the exception, i.e., $\emptyset = \text{first}(hs, \xi)$, then rule (\text{Catch}_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 in an inconsistent state. For example, an incoming exception may crash a thread inside a critical section and cause it to hold a lock in an inconsistent state. 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 $\xi$. If a handler for that exception is in the list, i.e., $h = \text{first}(hs, \xi)$, then (\text{Catch}_1) passes control to it. If no handler matches the exception, i.e., $\emptyset = \text{first}(hs, \xi)$, then rule (\text{Catch}_2) simply propagates the exception.

V. ASYNCHRONOUS EXCEPTIONS

\text{MAC}_\text{async} supports sending and handling asynchronous exceptions by means of two new primitives \text{throwTo} and \text{catch}, see Figure 9. Primitive \text{throwTo} $t \xi$ raises the exception $\xi$ of abstract type $\chi$ \textit{asynchronously} in the thread with identifier $t$. Intuitively, this operation constitutes a write effect, therefore \text{MAC}_\text{async} restricts its API according to the \texttt{no write-down} rule to enforce security. To this end, the API ensures that the security label of the receiver thread ($\ell_{hi}$) is at least as sensitive as the label of the sender ($\ell_i$) through the type constraint $\ell_i \sqsubseteq \ell_{hi}$. 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 \texttt{react} to asynchronous signals by handling exceptions appropriately. Primitive \text{catch} $t\ hs$ takes as a parameter a computation $t$ and a list containing pairs of exceptions and handlers. Then, if an exception $\xi$ is raised during the execution of $t$, the handler corresponding to the first exception matching $\xi$ in the list $hs$, if there is one, gets executed.

Figure 10 extends the calculus with value \textit{raise} $\xi$, which indicates that the computation is in an \textit{exceptional state}, and a new event \textit{throw}$(\xi, n)$, which instructs the runtime to

\begin{itemize}
\item \text{throwTo} :: $\ell_i \sqsubseteq \ell_{hi} \Rightarrow TId\ \ell_{hi} \rightarrow \chi \rightarrow \text{MAC}\ \ell_i\ ()$
\item \text{catch} :: $\text{MAC}\ \ell\ \tau \rightarrow [(\chi, \text{MAC}\ \ell\ \tau)] \rightarrow \text{MAC}\ \ell\ \tau$
\end{itemize}

\begin{itemize}
\item \text{mask} :: $\chi \rightarrow \text{MAC}\ \ell\ \tau \rightarrow \text{MAC}\ \ell\ \tau$
\item \text{unmask} :: $\chi \rightarrow \text{MAC}\ \ell\ \tau \rightarrow \text{MAC}\ \ell\ \tau$
\end{itemize}

Intuitively, primitive \text{mask} $\xi\ t$ runs computation $t$ with exceptions $\xi$ 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 $\xi$, and raised once the execution goes past the \text{mask} instruction. Term \text{unmask} $\xi\ t$. 

\footnote{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., \text{mask} $[\xi_1, \xi_2]$ $t$ behaves exactly like \text{mask} $\xi_1$ (\text{mask} $\xi_2$ $t$).}
works the other way around and enables exceptions $\xi$ 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 unmask $\xi$ (mask $\xi'$ $t$) receives exception $\xi$ while executing $t$, and if $\xi \neq \xi'$ and $t$ does not contain any mask $\xi$ instruction, then the exception gets raised, i.e., unmask $\xi$ (mask $\xi'$ (⋯ raise $\xi$ ⋯)).

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 $\xi_s$, on the left, and the list of remaining exceptions $\xi_s'$ 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 mask($M, \xi$) = $\lambda \xi' . \xi \equiv \xi' \lor M(\xi')$ and unmask($M, \xi$) (analogous). In particular, the rules reduce term mask $\xi$ $t$ (respectively unmask $\xi$ $t$) by executing term $t$ with modified mask $M_2$ obtained from disabling (enabling) exception $\xi$ in the current mask $M_1$, i.e., $M_2 =$ mask($M_1, \xi$) ($M_2 =$ unmask($M_1, \xi$)). When a nested, masked computation has completed, either successfully (return $t$) or not (raise $\xi'$), rules (MASK1) and (MASK$\xi$) simply propagate the result.

The masking context $M$ and the list of pending exceptions $\xi_s$ 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 ($\gg=$). (The rules for the other constructs are modified in a similar way, we refer the reader to the Agda mechanization for details).

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

Fig. 10: Syntax and semantics for asynchronous exceptions.

Fig. 12: Masking semantics of bind ($\gg=$).
In the elimination rules of the semantics, we apply function \( \text{unmasked}_|[M, \xi_s] \) to determine whether a pending exception should be raised. For example, if all exceptions are masked, i.e., \( \text{unmasked}_|[M, \xi_s] = \emptyset \), then rule (BIND2) steps as usual. In contrast, if an unmasked exception is pending, i.e., \( \text{unmasked}_|[M, \xi_s] = (\xi, \xi'_s) \), rule (BIND-RAISE) raises it, i.e., raise \( \xi \), and the thread steps with buffer \( \xi'_s \) where exception \( \xi \) has been removed.

Once raised, exceptions propagate unconditionally via rule (BIND), 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 \( t \) with the list of pending exceptions \( \xi_s \) 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 (THROW) processes event \( \text{throw}_\ell(\xi, n') \) by delivering exception \( \xi \) to the thread \( (t, \xi'_s, M') \) identified by \( (\ell', n') \). Since exceptions are processed in the same order as they are delivered, the rule inserts \( \xi \) at the end of the buffer \( \xi'_s \), i.e., \( \xi'_s \leftarrow \xi \).

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 \( \text{takeMVar} \), the rules for \( \text{putMVar} \) are symmetric. Rule (TAKE1) covers the case where no unmasked exception is pending, i.e., \( \text{unmasked}_|[M, \xi_s] = \emptyset \), and the thread can step because the variable is full, i.e., \( (t, n) \mapsto \langle t \rangle \in \Sigma \), and thus the rule returns its content \( t \) and empties the variable, i.e., \( \Sigma[(t, n) \mapsto \emptyset] \). On the other hand, in rule (TAKE-RAISE-UNMASKED), an unmasked exception is pending, i.e., \( \text{unmasked}_|[M, \xi_s] = (\xi, \xi'_s) \), thus, regardless of the variable being full or empty, i.e., \( (t, n) \mapsto c \in \Sigma \), the rule aborts the computation and raises the exception \( \xi \) without modifying the store. Lastly, in rule (TAKE-RAISE-MASKED), the variable is empty, i.e., \( (t, n) \mapsto \emptyset \in \Sigma \), and the thread should block and get stuck. However, an exception \( \xi \) is pending in the buffer \( \xi'_s \), i.e., \( \xi_s = \xi : \xi'_s \), therefore—regardless of the masking context \( M \)—the thread is resumed by raising exception \( \xi \). In the rule, the condition \( \text{unmasked}_|[M, \xi_s] = \emptyset \) 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 (TAKE-RAISE-UNMASKED) and raise the first unmasked exception, or via rule (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 (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 MACasync and that, we believe, can help programmers to write code that is more robust to asynchronous exceptions.

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

```
  do tidL ← forkL (do catch loop [(\xi, printL 1)])
      ← forkH (do s ← unlabel secret
               if s then throwToLeaky tidL \xi
           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. MACasync rejects such attacks by statically enforcing the no write-down rule in the API of \( \text{throwTo} \), which would make the code above ill-typed.

#### b) Asynchronous throwTo: In MACasync, primitive \( \text{throwTo} \) is itself an asynchronous operation. As rule (THWO2) in Figure 10 shows, primitive \( \text{throwTo} \) 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
in the GHC runtime provides only a synchronous version of \texttt{throwTo}. Would \texttt{MAC}_{async} be secure with a synchronous primitive \texttt{throwToSync}? No, unfortunately the possibility of two threads synchronizing by throwing exceptions opens a new covert channel. Consider the following example, where \texttt{throwToSync} has synchronous semantics, i.e., \texttt{throwToSync} blocks the sender thread until the exception is raised in the receiver thread.

\begin{verbatim}
  do tidH ← forkH (do s ← unlabel secret
      if s then mask ξ loop else loop)
    no_ops
  throwToSync tidH ξ
  printL 0
\end{verbatim}

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 \texttt{mask ξ loop}. After waiting for a sufficient amount of time through \texttt{no_ops}, the public thread sends exception \texttt{ξ} synchronously to the secret thread. If the secret thread is looping in the masked block, the exception \texttt{ξ} will never be raised, causing the public thread to block forever on \texttt{throwToSync} and thus suppressing the final public output \texttt{printL 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 \texttt{MVar}, synchronous communication primitives perform both read and write side-effects, therefore \texttt{throwToSync} cannot operate securely between threads at different security levels. Even though Concurrent Haskell provides only the equivalent of \texttt{throwToSync}, we can still derive a secure asynchronous implementation for \texttt{throwTo} by internally forking an isolated thread that calls the unsafe \texttt{throwToSync}, i.e., we define \texttt{throwTo t ξ} as \texttt{fork (throwToSync t ξ) return ()}.

\textbf{c) Reliable exception delivery:} In \texttt{MAC}_{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.

\textbf{D. Relation to MAC}

\texttt{MAC}_{async} extends \texttt{MAC} with asynchronous exceptions \[23\]. \texttt{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 \texttt{MAC}, threads can communicate with each other only synchronously and \textit{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 \texttt{MAC}_{async} can communicate \textit{directly}, by sending exceptions to the identifier of the intended receiver thread, and also to threads at a different, more sensitive security level. \texttt{MAC}_{async} leverages the mechanization of \texttt{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.

\section{VI. Security Guarantees}

This section shows that \texttt{MAC}_{async} satisfies progress-sensitive non-interference (PSNI). We begin by describing our proof technique based on \textit{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.

\textbf{A. Term Erasure}

\textit{Term erasure} is a widely used technique to prove non-interference properties of IFC languages (e.g. \[24\] \[26\] \[30\] \[34\] \[37\] \[38\] \[39\]). The technique takes its name from the \textit{erasure function}, which removes secret data \textit{syntactically} from program terms. To this end, the erasure function, written \(\varepsilon_{L_A}(t),\)

\begin{verbatim}
https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Exception.html#v:throwTo
\end{verbatim}
rewrites the sub-terms of \( t \) above the attacker’s security level \( \ell_A \) 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 \( \varepsilon_t(t) \) along the orange path leads to the same term \( \varepsilon_t(t') \) 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 \( \ell_A \) would remain in the erased term \( \varepsilon_t(t') \), 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 configurations, stores, thread pools, and thread states in Figure 17. Configurations are erased component-wise, i.e., \( \varepsilon_{\ell_A}(\Sigma, \Theta) = (\varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Theta)) \). Thread pools \( \Theta \) containing secret threads are entirely removed by the erasure function, i.e., \( \varepsilon_{\ell_A}(\Theta(t)) = \bullet \) if \( \ell \not\sqsubseteq \ell_A \), while those containing thread pools are erased homomorphically, i.e., \( \varepsilon_{\ell_A}(\Theta(t)) = \varepsilon_{\ell_A}(\Theta(t)) \) if \( \ell \sqsubseteq \ell_A \), where \( \varepsilon_{\ell_A}([]) = [] \) and \( \varepsilon_{\ell_A}(T_s) = (\varepsilon_{\ell_A}(T_s), \varepsilon_{\ell_A}(T_s)) \). (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 \( (\bullet, [], \lambda_\bot, \text{False}) \) 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 \code{throwTo} to communicate with a secret thread gets stuck, since rule \( \text{(THROW)} \) would try to deliver an exception into thread pool \( \bullet \). To recover from this situation, we apply the two-step erasure technique. This technique re-establishes the core simulation property fundamental for security (Fig. 15). For example, term \( \text{throwTo}\bullet \ t \) \( \xi \) reduces just like \code{throwTo} in rules \( \text{(THROW1)} \) and \( \text{(THROW2)} \), thus respecting the simulation property of the sequential semantics. However, instead of generating a regular event \code{throwTo}(\( \xi \), \( n \)), which would get the concurrent configuration stuck in rule \( \text{(THROW)} \), it generates a new event \code{throwTo}(\( \xi \), \( n \)). 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).

### 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 \( c_1 \rightsquigarrow c_2 \) and \( c_2 \rightsquigarrow c_3 \), then \( c_2 \equiv c_3 \).

The symbol \( \equiv \) above denotes syntactic equality up to alpha-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 \( \ell \) of the thread is below the level of the attacker, i.e. \( \ell \subseteq \ell_A \), 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., \( \ell \nsubseteq \ell_A \), then the configurations before and after the step are indistinguishable to the attacker. This indistinguishability relation is called \( \ell_A \)-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 \( \ell, n \vdash c_1 \rightsquigarrow c_1' \) then

1. \( \ell, n \vdash \varepsilon_{\ell_A}(c_1) \rightsquigarrow \varepsilon_{\ell_A}(c_1') \), if \( \ell \subseteq \ell_A \), or
2. \( c_1 \approx_{\ell_A} c_1' \), if \( \ell \nsubseteq \ell_A \).

Using Lemmas 1 and 2 we prove progress-sensitive non-interference (PSNI), where symbol \( \rightsquigarrow^* \) denotes the transitive reflexive closure of \( \rightsquigarrow \) 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 \( \ell, n \vdash c_1 \rightsquigarrow c_1' \), then there exists a configuration \( c_2' \) such that \( c_1' \approx_{\ell_A} c_2' \) and \( c_2 \rightsquigarrow^* c_2' \).

Proof. By cases on \( \ell \subseteq \ell_A \).

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

If \( \ell \nsubseteq \ell_A \), 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 \( \ell_A \)-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 MACasync. 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 MACasync. 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 MACasync could represent a basic building block to enforce them reliably.

Zee is an IFC language for implementing secure (timingsensitive) 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 MACasync 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 MACasync in 3,000 lines of Agda and proved progress-sensitive non-interference.

As future work, we plan to use MACasync 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 MACasync to build realistic systems. For instance, we expect MACasync 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


