Securing Verified IO Programs Against Unverified Code in F*

Cezar-Constantin Andrici, Ștefan Ciobâcă, Cătălin Hriţcu, Guido Martínez, Exequiel Rivas, Éric Tanter, Théo Winterhalter

Securing Verified IO Programs Against Unverified Code in F*

CEZAR-CONSTANTIN ANDRICI*, MPI-SP, Germany
ȘTEFAN CIOBÂCĂ, Alexandru Ioan Cuza University, Romania
CĂTĂLIN HRIȚCU, MPI-SP, Germany
GUIDO MARTÍNEZ, Microsoft Research, USA
EXEQUIEL RIVAS, Tallinn University of Technology, Estonia
ÉRIC TANTER, University of Chile, Chile
THÉO WINTERHALTER, Inria Saclay, France

We introduce SCIO*, a formally secure compilation framework for statically verified programs performing input-output (IO). The source language is an F* subset in which a verified program interacts with its IO-performing context via a higher-order interface that includes refinement types as well as pre- and post-conditions about past IO events. The target language is a smaller F* subset in which the compiled program is linked with an adversarial context that has an interface without refinement types, pre-conditions, or concrete post-conditions. To bridge this interface gap and make compilation and linking secure we propose a formally verified combination of higher-order contracts and reference monitoring for recording and controlling IO operations. Compilation uses contracts to convert the logical assumptions the program makes about the context into dynamic checks on each context-program boundary crossing. These boundary checks can depend on information about past IO events stored in the state of the monitor. But these checks cannot stop the adversarial target context *before* it performs dangerous IO operations. Therefore linking in SCIO* additionally forces the context to perform all IO actions via a secure IO library, which uses reference monitoring to dynamically enforce an access control policy before each IO operation. We prove in F* that SCIO* soundly enforces a global trace property for the compiled verified program linked with the untrusted context. Moreover, we prove in F* that SCIO* satisfies by construction Robust Relational Hyperproperty Preservation, a very strong secure compilation criterion. Finally, we illustrate SCIO* at work on a simple web server example.

CCS Concepts: • **Software and its engineering** → **Software verification**; **Compilers**.

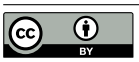
Additional Key Words and Phrases: secure compilation, formal verification, proof assistants, input-output

1 INTRODUCTION

Increasingly realistic programs have been written and verified in proof-oriented programming languages like Coq, Isabelle/HOL, Dafny, and F* for obtaining strong static guarantees of correctness and security [Appel 2016; Arasu et al. 2023; Bhargavan et al. 2021a; Delignat-Lavaud et al. 2017; Gu et al. 2016; Hance et al. 2020; Ho et al. 2022; Klein et al. 2010; Leroy 2009; Li et al. 2022; Murray et al. 2013; Protzenko et al. 2020; Ramananandro et al. 2019; Zakowski et al. 2021; Zinzindohoué et al. 2017]. One way to speed up the development process in such languages is to use existing unverified libraries written in more mainstream languages. In such cases, the program is usually verified under some *assumptions* about the libraries, and then the verified program is compiled and

*First author.

Authors' addresses: Cezar-Constantin Andrici, MPI-SP, Bochum, Germany, cezar.andrici@mpi-sp.org; Ștefan Ciobâcă, Alexandru Ioan Cuza University, Iași, Romania, stefan.ciobaca@gmail.com; Cătălin Hriţcu, MPI-SP, Bochum, Germany, catalin.hritcu@mpi-sp.org; Guido Martínez, Microsoft Research, Redmond, WA, USA, guimartinez@microsoft.com; Exequiel Rivas, Tallinn University of Technology, Estonia, exequiel.rivas@ttu.ee; Éric Tanter, University of Chile, Computer Science Department, Santiago, Chile, etanter@dcc.uchile.cl; Théo Winterhalter, Inria Saclay, France, theo.winterhalter@inria.fr.



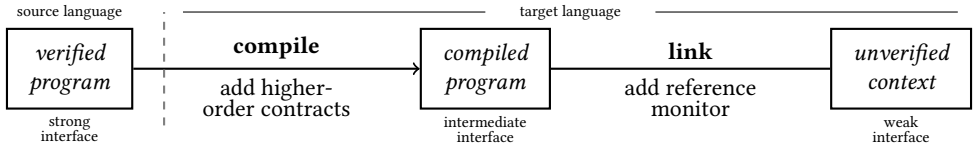


Fig. 1. An overview of SCIO*.

linked with the unverified libraries. The assumptions about the linked libraries can be expressed explicitly, as specifications [Cok and Leino 2022], or implicitly—i.e., assuming that the libraries respect the abstractions of the language of the verified program and e.g., do not have undesirable side effects. In either case, these assumptions are necessary for the verification of any global property characterizing the behavior of the program together with the libraries. However, current compilers from proof-oriented languages (e.g., from Coq and F* to OCaml [Letouzey 2008; Sozeau et al. 2020] or C [Anand et al. 2017; Paraskevopoulou et al. 2021; Protzenko et al. 2017], or from Dafny to C#, or from F* and Dafny to Assembly [Bond et al. 2017; Fromherz et al. 2019]) take for granted that the linked unverified code cannot break the assumptions on which the verified program relies. This is unsound and thus provides no global guarantees after compilation and linking. More importantly, this is insecure if—as we assume in this paper—the linked code is *untrusted* (e.g., vulnerable, compromised, or malicious).

To ensure security, one could statically verify the linked code (e.g., library) to prove that it cannot be compromised and all the assumptions we make about it hold [Guéneau et al. 2023; Patterson et al. 2022; Rao et al. 2023; Sammler et al. 2023], but this would likely require much effort, as static verification most often involves user interaction and expertise, thus taking away the simplicity of just using the unverified code. The alternative is to enforce the assumptions at runtime by converting them into dynamic checks [Protzenko et al. 2020; Zinzindohoué et al. 2017]. In this paper we assume that one is willing to (re)write the linked code in the proof-oriented language, so that all assumptions are explicit. However, to speed up development, one wants to enforce the explicit assumptions about linked unverified code dynamically instead of proving them statically.

In this paper we introduce SCIO*, which provides a *formally verified* way to systematically convert into dynamic checks all the assumptions made by verified F* [Swamy et al. 2016] programs with IO about linked unverified F* code with IO—e.g., reading and writing files and network sockets. We look at this problem through the lens of secure compilation [Abate et al. 2019; Patrignani et al. 2019]: **SCIO* is a formally secure compilation framework for IO programs** consisting of a compiler and a linker between two languages shallowly embedded in F*.

The *source language* is an F* subset in which a verified *partial source program* interacts with a *source context* via a *strong higher-order interface*, consisting of specifications expressed as refinement types and pre- and post-conditions. Refinement types are used to constrain the values of a base type with a logical formula (e.g., the value of an integer is larger than zero). The pre- and post-conditions of a function can depend on its arguments, can specify its result, and can also specify its IO behavior by considering the trace of past IO events—each time an IO operation is performed, an event containing the arguments and the result is appended to the trace. A pre-condition can constrain the trace of past IO events at the time the function is called (e.g., it could require that a file is currently open by looking back in the trace for an open event for the corresponding file descriptor and making sure that no close event happened afterwards for this descriptor). A post-condition can additionally take into account the IO events produced by the function itself when it returns (e.g., the function closed all file descriptors that it opened) and can also specify the relation between the result value and the return-time trace (e.g., the returned value was read from a file).

The *target language* is a smaller F* subset in which the compiled program is linked with an adversarial *target context* that has a *weak interface* without refinement types, pre-conditions, or

concrete post-conditions (see Figure 1). The partial source program and the target context can only interoperate securely when their interfaces match, which is not the case in our setting because the strong interface of the verified program contains concrete specifications expressed as refinement types and pre- and post-conditions, while the weak interface of the context does not.

SCIO⋆ enables a verified partial source program to be securely compiled and linked against an arbitrary target context. SCIO⋆ supports both the case in which the untrusted context is a library used by the program and the case in which the program is a library used by the untrusted context. To achieve this secure interoperability SCIO⋆ uses a combination of (1) *reference monitoring* [Anderson 1973], introduced by the linker, and (2) *higher-order contracts* [Findler and Felleisen 2002], introduced by the compiler. These techniques have, as far as we know, not been applied so far to our setting, where we are protecting statically verified code from unverified code, and where we aim for strong formal security guarantees. We use reference monitoring and higher-order contracts to bridge the gap between the strong interface of the source program and the weak interface of the untrusted target context by giving the compiled program an *intermediate interface* in between (see Figure 1).

(1) The reference monitor records in its state information about the trace of IO events that happened so far during the execution. The monitor uses this information to enforce an *access control policy* by performing a dynamic check before each IO operation of the context (e.g., preventing access to a file descriptor associated with a password file or network socket). SCIO⋆ implements this by linking the target context with a secure IO library that dynamically enforces the policy. This policy enforcement on each IO operation of the context is necessary, because checking some post-conditions when the context returns would be too late to prevent bad IO events from happening. For example, a post-condition of the context could state that it has neither accessed the passwords file, nor the network. If we only detect a violation of this policy after the context returns, the damage has already been done, with passwords leaked over the network. Instead, we use the monitor to prevent illegal IO events from actually happening, and the trace is then unaffected as we do not record such prevented events. However, reference monitoring is not enough on its own, since the strong interface contains refinement types and some pre- and post-conditions that cannot be enforced at the level of IO operations, but have to be enforced on the higher-order boundary between the program and the context (e.g., the pre-condition of a callback sent to the context can require that its argument is an open file descriptor).

(2) Higher-order contracts are used by SCIO⋆ during compilation to convert the assumptions on the boundary between the partial program and the context into dynamic checks. The SCIO⋆ compiler wraps the program in new functions with weaker types and adds dynamic checks before each function call and after each function return crossing the boundary between program and context. This dynamically enforces refinement types on arguments and results, as well as pre- and some post-conditions of functions. To enforce pre- and post-conditions related to IO behavior, the higher-order contracts access the state of the monitor to get information about the IO events that happened before the function was called and during the execution of the function.

We are the first to secure verified IO programs against unverified code, and moreover we provide strong machine-checked security guarantees. For this we make the following **contributions**:

- We introduce SCIO⋆, an F⋆ framework for securely compiling a verified partial IO program and linking it against an IO context with a weak higher-order interface. We bridge the interface gap between the verified program and the untrusted context by using a combination of higher-order contracts and a reference monitor, which share state that records information about prior IO events. Moreover, SCIO⋆ takes advantage of the program being statically verified and performs no dynamic checks when the program performs IO or when it passes control to the context.
- We statically verify that SCIO⋆ adds enough dynamic checks to guarantee that the program and the context can interoperate securely. This verification includes the implementation of

higher-order contracts, the wrapping of the context's IO library with the monitor's access control checks, as well as their combination. Moreover, we prove in F* the soundness of the entire SCIO* framework, showing that a global trace property holds for the compiled program linked with the untrusted context. This proof is done modularly by typing and also heavily benefits from F*'s SMT automation.

- We represent computations in our shallowly embedded source and target languages by using a new monadic effect MIO—i.e., Monitored IO—that is at its core a way to statically verify terminating IO programs, engineered to take advantage of SMT automation in F*. In addition to usual IO operations (e.g., reading and writing files and network sockets), MIO contains a get_mstate operation, which models in a simple abstract way access to the reference monitor state, and which allows us to implement the dynamic checks done by the monitor and higher-order contracts. In addition, at the specification level we distinguish between events produced by the program and those produced by the context, which enables enforcing a stronger specification on the untrusted context.
- To model in a simple way that our shallowly embedded contexts cannot directly access the IO operations and the monitor's internal state we propose a novel use of flag-based effect polymorphism. The flag is an index of the MIO monad that controls which operations a computation can access. Flag-based effect polymorphism is necessary since F* gives up on more general kinds of effect polymorphism in order to gain from SMT automation. In addition to this shallow embedding of contexts, we introduce a syntactic representation of target contexts in a small deeply embedded language that can be translated into the shallow embedding.
- We show that SCIO* is secure by providing a machine-checked proof in F* that it satisfies Robust Relational Hyperproperty Preservation (RrHP), which is the strongest secure compilation criterion of Abate et al. [2019], and in particular stronger than full abstraction. Intuitively this ensures that SCIO* provides enough protection to the compiled program so that linked target contexts do not enjoy more attack power than a source context would have against the original source program. While proofs of such criteria are generally challenging [Abate et al. 2019; Devriese et al. 2017; Jacobs et al. 2022; New et al. 2016; Patrignani et al. 2019], we have carefully set things up in SCIO* so that our proof is simple by construction, in particular because: (1) our languages are shallowly embedded in F*; (2) we used flag-based effect polymorphism to model the context; (3) we designed our higher-order contracts mechanism so that we can define both compilation and back-translation in a way that they satisfy a syntactic inversion law that immediately implies RrHP (i.e., compiling the program and linking it with the context is syntactically equal to back-translating the context and linking it with the program). We prove RrHP for both the case in which the context is a library used by the program and the case in which the program is a library used by the context.
- We illustrate SCIO* on a main case study in which we statically verify a simple web server in F*, compile it to our target language, and link it against some adversarial request handlers and a non-adversarial request handler serving files. This illustrates that our languages are expressive enough to write interesting code and that the MIO monadic effect enables effective verification.

Outline. We start by illustrating the key ideas of SCIO* on the verified web server case study (§2). We then present the MIO effect (§3) and the implementation of higher-order contracts (§4). We put these pieces together to define SCIO* and prove it soundly enforces a global safety property and it satisfies RrHP (§6). Next, we explain how we execute the web server in OCaml (§7) and illustrate a few more examples (§8). Finally, we discuss related (§9) and future work (§10).

Longer-term goal. Achieving strong secure compilation criteria such as RrHP is extremely difficult, and no realistic compilation chain that achieves such criteria has ever been built [Abate et al. 2019; Patrignani et al. 2019]. We see our work as an important step towards building a formally

secure compilation chain from F⋆ to a safe subset of OCaml. This is a formidable research challenge though, so we carefully limited the scope of the current paper by focusing on IO as the single side effect and by assuming for now that the unverified context is also written in a shallowly embedded subset of F⋆, but only respects a weak interface. In §10 we discuss how these assumptions could be lifted in the future to achieve formally secure compilation from F⋆ to OCaml.

2 SCIO⋆ IN ACTION

```
type req_handler =
  (client:file_descr) →
  (req:buffer{valid_http_request req}) →
  (send:(res:buffer{valid_http_response res} → MIO (either unit err)
                                                 (requires (λ h → did_not_respond h))
                                                 (ensures (λ _ _ lt → ∃r. lt = [EWrite _ client r]))) →
  MIO (either unit err) (requires (λ h → did_not_respond h))
                        (ensures (λ h r lt → (wrote_to client lt ∨ Inr? r) ∧
                                             handler_only_opens_and_reads_files_from_folder lt ∧
                                             web_server_only_writes lt)
let web_server (handler:req_handler) :
    MIO unit (requires (λ h → ⊤)) (ensures (λ _ _ lt → every_request_gets_a_response lt) =
  let s = socket () in setsockopt s SO_REUSEADDR true; bind s "0.0.0.0" 3000; listen s 5; ...
  let client = select_client s in
  let req = get_req client in
  if Inr? (handler client req (write client)) then sendError 400 client;
  close client; ...
```

Fig. 2. SCIO⋆ case study: web server that takes a request handler as argument. The types are simplified.

We illustrate the key ideas of SCIO⋆ using our main case study as a running example:[1] the partial program is a simple web server that is verified in F⋆ to respond to every request, while the unverified context is a request handler represented as a higher-order function. The web server has the initial control and it gets the request handler as an argument. We carefully crafted the case study to be higher-order and to contain the interesting types of specifications SCIO⋆ supports. The case study is still moderately realistic, since we can run our web server and it can handle HTTP requests from a real browser (§7). We also applied SCIO⋆ to a few other examples we discuss in §8.

We start with the strong interface of the web server, how it is verified (§2.1), and the assumptions that it makes about the handler, which is our adversarial target context (§2.2). We then explain how the intermediate interface bridges the gap between the web server and the handler (§2.3), before presenting the weak interface of the handler (§2.4). We then introduce the executable checks and access control policy the SCIO⋆ requires to enforce the specification (§2.5). We use the checks to weaken the assumptions the web server makes about the untrusted handler by using higher-order contracts (§2.6). Finally, we strengthen the type of a target handler using reference monitoring to enforce the policy (§2.7).

[1] F⋆ syntax is similar to OCaml (val, let, match, etc). Binding occurrences b take the form x:t or #x:t for an implicit argument. We omit the type in a binding when it can be inferred. Lambda abstractions are written $\lambda b_1 ... b_n \rightarrow t$ (where t ranges over both types and terms), whereas $b_1 \rightarrow ... \rightarrow b_n \rightarrow C$ denotes a curried function type with result C, a computation type describing the effect, result, and specification of the function. Contiguous binders of the same type may be written ($v_1 ... v_n$: t). Refinement types are written b{t} (e.g., x:int{x≥ 0} represents natural numbers). Dependent pairs are written as $x:t_1 \& t2$. The squash t type is defined as the refinement _:unit{t}, and can be seen as the type of computationally-irrelevant proofs of t. For non-dependent function types, we omit the name in the argument binding, e.g., type #a:Type → (#m #n : nat) → vec a m → vec a n → vec a (m+n) represents the type of the append function on vectors, where both unnamed explicit arguments and the return type depend on the implicit arguments. A type-class constraint {| d : c t1 .. tn |} is a special kind of implicit argument, solved by a tactic during elaboration. Type0 is the lowest universe of F⋆; we also use it to write propositions, including ⊤ (True) and ⊥ (False). We generally omit universe annotations. The type either t_1 t_2 has two constructors, Inl:(#t_1#t_2:Type) → t_1 → either t_1 t_2 and Inr:(#t_1#t_2:Type) → t_2 → either t_1 t_2. Expression Inr? x tests whether x is of the shape Inr y. Binary functions can be made infix by using backticks: x `op` y stands for op x y.

2.1 The verified partial program (web server)

Figure 2 illustrates the most important part of the web server's implementation. It starts by opening a TCP socket (line 13) and then, inside an elided terminating loop,[2] it accepts clients and waits for incoming data in a non-blocking way. The web server waits for requests from multiple clients at the same time. For each client that sends data (line 14), the web server reads it and validates the HTTP request (line 15), and then it passes the request to the handler (line 16). If the handler failed to respond to the request—i.e., it returned an error value (tagged with Inr)—then the web server responds with an error (400) to the client.

The web server takes as argument a handler of type req_handler and its result type (line 12) is that of an MIO computation that returns a unit and has a trivial pre-condition (indicated by requires) and a post-condition (indicated by ensures). The pre- and the post-condition are part of the type, as indices of the MIO monadic effect. Similarly, the type req_handler also contains many interesting refinement types and pre- and post-conditions. We highlight the following specifications that appear on the types of the web server and handler:

(1) The post-condition of the web_server (line 12) ensures that it responds to all accepted clients.
(2) The first two arguments of the handler are a file descriptor client and a buffer req that is guaranteed to contain a valid HTTP request thanks to the refinement type (line 3). The handler also has as pre-condition that no response has been sent yet to the latest request (line 7).
(3) The third argument of the handler is a callback send (lines 4-6) that expects as argument a buffer that contains a valid HTTP response and requires that no response has been sent yet to the latest request—i.e., the same pre-condition as the handler. The concrete callback for send is passed by the web server and it simply writes the response to the client.
(4) The type of the handler ensures that either it wrote to the file descriptor it got as argument or it otherwise returns an error value (tagged with Inr, line 8).
(5) The type of the handler also ensures that it only opens files from a specific folder, reads only from its own opened files, and closes only them (line 9). It also ensures that during its execution only the trusted web_server can write (so only when the handler calls the send callback, line 10). Any other IO operation of the handler or the web_server is not permitted by the post-condition.

These specifications describe how the web server behaves and also what assumptions the web server makes about the request handler. The assumptions about the request handler are necessary to verify that the web server satisfies its post-condition. In particular, without the post-condition of the handler ensuring that it writes to the client when it returns a success value (tagged with Inl), one would not be able to verify that the web server responds to every request. The post-condition of the web server is defined using the following predicate that checks whether every request (read from a file descriptor) is followed at some point by a response (write to the same file descriptor).

```
let every_request_gets_a_response (lt:trace) : Type0 =
  let rec aux = (λ lt read_fds →
    match lt with
    | [] → read_fds == []
    | ERead (fd, _) (Inl _) :: tl → aux tl (fd :: read_fds)
    | EWrite (fd, _) _ :: tl → aux tl (filter (λ fd' → fd ≠ fd') read_fds)
    | _ :: tl → aux tl read_fds) in
  aux lt []
```

The proof that the web server satisfies its post-condition is done mostly automatically with the help of the SMT solver. The implementation is split into several functions, which makes verification more modular by feeding smaller verification conditions to the SMT solver. We needed to state and prove a few lemmas (some using interactive proofs [Martínez et al. 2019]), about the

[2] Our framework supports recursion, as long as F* can prove termination. General recursion is future work though (§10).

every_request_gets_a_response predicate, which had to be proven by induction on the trace, something that the SMT solver cannot do on its own. However, once these lemmas were proven, the SMT solver was able to exploit them to prove the specifications of the various parts of our web server without further manual intervention. This is thanks to the SMT automation F* provides for monadic effects and to our careful design of the MIO monadic effect (§3).

2.2 The assumptions about the context (handler)

The strong interface of the web server includes the type req_handler, which defines the expected specification of the handler (the other components of a strong interface are given in §6.1 but they are not yet relevant here). A traditional compiler that just erases specifications[3] would for instance convert req_handler into a type without refinement types and without pre- and post-conditions:

```
type too_weak_handler_type =
  file_descr → buffer → (buffer → MIO (either unit err)) → MIO (either unit err)
```

However, it would be unsound and insecure to directly link an arbitrary inhabitant of this type to the partial program. The simplest example of an adversarial handler that breaks soundness is the one that immediately returns a success value. This breaks specification 4 of the web server that expects the handler to write to the client at least once when it returns a success value.

```
let adversarial_handler1 client req send = Inl ()
```

To securely compile the web server, it is required to add dynamic checks to enforce the assumptions about the context—i.e., specifications 3, 4 and 5. SCIO* enforces specification 3 and 4 using higher-order contracts and enforces specification 5 using an access control policy. The post-condition of the handler (4-5) is the most interesting because it is enforced in part by higher-order contracts and in part by reference monitoring. The first part of the post-condition—i.e., checking whether the handler wrote to the client (4)—can be soundly enforced by a contract when the handler returns, but not using reference monitoring at the level of the IO operations. The second part of the post-condition 5 specifies the IO behavior of the handler, which cannot be securely checked by a contract when the handler returns. The post-condition 5 for instance specifies that the handler does not open files outside of a specific folder: if the untrusted handler opens a password file from a different folder we could detect that when it gives back control to the web server; however, this still breaks our post-condition since the bad event has already happened and it is too late to do anything about it. Therefore, we prevent the violation from happening by using reference monitoring—i.e., using an access control policy that enforces the two predicates of specification 5 (more in §2.7).

Because the web server is statically verified, we do not have to dynamically enforce anything when the server passes control to the handler—i.e., the pre-condition of the handler, or that the web server passes a valid HTTP request to the handler (2). This also includes when control *returns* from the web server into the handler, such as when the send callback returns after being invoked by the handler—no dynamic enforcement is needed because send is verified to satisfy its post-condition.

2.3 Bridging the gap with the intermediate interface

SCIO* combines reference monitoring for recording and controlling IO events together with higher-order contracts that have access to the monitor state. Together they bridge the gap between the strong interface of the verified program and the weak interface of the untrusted context—e.g., between the req_handler type and something like the too_weak_handler_type with erased specifications

[3]As mentioned in §1, current extraction mechanisms from proof-oriented languages [daf 2023; Bond et al. 2017; Fromherz et al. 2019; Letouzey 2008; Protzenko et al. 2017] just erase specifications, even the extraction mechanisms that are formally verified to be correct [Anand et al. 2017; Paraskevopoulou et al. 2021; Sozeau et al. 2020].

above. The bridging produces a middle point we call the *intermediate interface* (Figure 1), which both the compiled partial program and the monitored target context share.

The meeting point between what our higher-order contracts do not enforce and what the reference monitor enforces is the access control policy. Therefore, an intermediate interface contains types annotated with no specifications except that each function has a post-condition stating that it satisfies the access control policy. We denote this by using the short notation MIO a ⊤ Σ, where ⊤ says that there is no pre-condition and Σ is the specification of the access control policy encoded as a post-condition. For instance, the intermediate interface version of req_handler looks as follows:

```
type intermediate_handler_type =
  file_descr→ buffer→ (buffer→ MIO (either unit err) ⊤ Σ)→ MIO (either unit err) ⊤ Σ
```

We define compilation as converting a program with a strong interface into a program with this kind of intermediate interface. Therefore, the compilation of the web server produces a function that expects a request handler of intermediate_handler_type. The Σ in this type is presented in §2.7—e.g., it prevents the handler from opening files outside a specific folder.

The target context gets an intermediate interface during target linking, when the reference monitor is added, because the monitor enforces that the context satisfies the access control policy. Since monitoring is done at the level of IO operations, to monitor the target context it is enough to link it with a secure IO library that dynamically enforces the access control policy. On the other hand, the compiled program can call directly the default IO operations that only record the IO events in the monitor state, but performs no dynamic checks.

2.4 The weak interface of the unverified context (handler)

The context gets its specification from the reference monitor, which enforces the access control policy for the IO operations. Therefore, we make explicit the dependency of the unverified context on the IO library by modeling the unverified context as being *parametric* over the library and also over the abstract access control policy enforced by the library. This definition captures the intuition that "the IO library gives specification to the context" because the abstract specification of the context is given by what IO operations the context uses, and if it uses only operations that enforce an abstract access control policy, then we can show that the context satisfies the abstract policy. So intuitively, the type of the unverified request handler looks more like this in SCIO*:

```
type still_too_weak_handler_type = Σ:erased _ → io_lib Σ →
                    file_descr → buffer → (buffer → MIO (either unit err) ⊤ Σ)) → MIO (either unit err) ⊤ Σ
```

where io_lib Σ is the type of the IO libraries that enforce a policy that has abstract specification Σ.

The weak interface clarifies that the abstract specification of the context depends on the specification enforced by the secure IO library. The fact that an abstract specification is still part of the weak interface does not invalidate the intuition that the context is unverified because the weak interface is parametric in this specification. One cannot verify any concrete specification when type-checking the context's code, but it is trivial to show that it inherits the abstract specification of the secure IO library. We also have further evidence that our weak interfaces are expressive enough to represent unverified contexts: (1) the implementation of our handlers, including the larger non-adversarial one that serves files; (2) a syntactic representation of simply-typed target contexts and a translation from any syntactic expression to our shallow embedding (see §6.6).

The benefit of defining weak interfaces like this in SCIO* is that after instantiation the monitored context has a specification in its type, which greatly simplifies the soundness proof of our secure compilation chain (§6.3).

This type would, however, not prevent the handler from directly calling the IO operations provided by the MIO monadic effect. In proof-oriented languages like Coq one could simply quantify

over the monadic effect [Maillard et al. 2019]—i.e., a computation monad indexed by a specification monad (see §3.1 for details)—and use such effect polymorphism to prevent access to IO operations. However, F⋆ doesn't allow quantifying over effects because it needs the concrete implementation of the specification monad to generate verification conditions for the SMT solver. Therefore, we were inspired from work by Brachthäuser et al. [2020] on *parametric effect polymorphism* and combined their idea with the idea of indexing a monadic effect with a flag [Rastogi et al. 2021].

So, for a start, we added a flag index to the MIO effect. A flag value is simply an inhabitant of a variant type tflag with four constructors: NoOps, GetMStateOps, IOOps and AllOps. By indexing a computation with an element of this type we can restrict the operations that the computation can perform: (1) NoOps means that no operation can be used in the computation, (2) GetMStateOps means that only get_mstate operation can be used, (3) IOOps means that only IO operations (open, read, etc.) can be used, but not get_mstate (4) AllOps means that the computation can access any of the operations. Because the target context is parametric in the flag, it prevents calls to any of the operations as the flag could take the value NoOps. We refer to this form of effect polymorphism as *flag-based effect polymorphism*. So the actual weak type of an unverified handler looks like this:

```
type weak_handler_type = fl:erased tflag → Σ:erased _ → io_lib Σ fl →
                         file_descr → buffer → (buffer → MIO (either unit err) fl ⊤ Σ)) → MIO (either unit err) fl ⊤ Σ
```

where fl is the flag (marked as erased so only usable at the specification level) which is also an index of io_lib. The instantiation of the target context happens during target linking, where the flag- and specification-polymorphic context is passed the AllOps flag, the concrete specification of an access control policy, and the secure IO operations that enforce the concrete access control policy.

The use of flag-based effect polymorphism is key to correctly model the attack capabilities of the source and target contexts, enabling us to achieve and prove RrHP (§6.4).

2.5 Providing the dynamic checks and the access control policy

Before discussing about how the dynamic checks are enforced, we first note that since some F⋆ specifications are not directly executable, SCIO⋆ requires the dynamic checks to be used inside the higher-order contracts and the access control policy to be enforced by the reference monitor. The F⋆ type checker asks for the necessary dynamic checks, and it also verifies that they indeed imply the specification. Since this verification is done in F⋆ it takes advantage of its support for SMT automation and interactive proofs [Martínez et al. 2019]. SCIO⋆ also leverages type classes in our implementation of higher-order contracts which gives more automation in figuring out the dynamic checks, which works great on simpler cases. While one can make arbitrary assumptions about the context, not all of them have sound dynamic checks that are both *precise enough* and *efficiently enforceable*, thus it becomes a design decision on what assumptions are made and what dynamic checks are picked. For our example, SCIO⋆ requires dynamic checks that imply the pre-condition of send (3) and the first part of the post-condition of the handler (4), as well as an access control policy that implies the other part (5).

While our specifications use traces, these can be impractical for dynamic enforcement because they grow indefinitely. Instead, SCIO⋆ gives the ability to pick a different state for the monitor as long as it abstracts the trace, and efficient custom checks over this abstract state that don't have to traverse a linear trace. We discuss more about how we do this in §5, but until then, in what follows we assume that the monitor stores the trace of events to simplify the technical details.

2.6 Compilation using higher-order contracts

The SCIO⋆ compiler converts a program with a strong interface into a program with an intermediate interface. For example, compilation of the web server wraps it into a new function that takes as argument an intermediately-typed handler. The intermediately-typed handler then is made

strongly-typed (of type req_handler) by enforcing higher-order contracts on it. SCIO* implements higher-order contracts by using two dual functions: export converts strongly-typed values into intermediately-typed values, and import converts intermediately-typed values into strongly-typed values. The import and export functions are based on the types of the values and defined in terms of each other, which is needed for supporting higher-order functions [Findler and Felleisen 2002].

Import of the intermediately-typed handler wraps it inside of a new function of type req_handler, in which first the strongly-typed arguments are exported. The most interesting is the export of send: export wraps it into a new intermediately-typed function that enforces the refinement type and send's pre-condition (3). Not enforcing the refinement or the pre-condition of send would be unsound, situation in which F* would not even let us call send, since it would enable the handler to respond to the same client more than once or to call it with an invalid HTTP response:

```
let adversarial_handler2 client req send = send (Bytes.of_string "hello") // invalid HTTP response
```

In case the dynamic checks of the refinement and of the pre-condition succeed, the send function is called and its result is exported and returned, otherwise if one of the check fails then Inr Contract_failure is returned. This is why the return type of send is either unit err, meaning that it can fail. Thus, when a contract fails (or and when the monitor prevents a bad IO operation), an error is returned, which allows the program and the context to handle the errors.[4]

After exporting the strongly-typed arguments, the intermediately-typed handler is called with them and an intermediately-typed result is obtained. After importing the result, but before returning it, we have to dynamically enforce the post-condition—i.e., that it wrote to the client. This dynamic check prevents the previously presented adversarial_handler1 attack to work because after being imported, instead of returning Inl (), it returns the error Contract_failure, thus falling into the branch of the web server that checks whether the handler ran into an error and responds with HTTP error 400, ensuring that every request gets a response (1).

Since SCIO* gets a proof that enforcing this dynamic check implies the post-condition 4 (§2.5) and since the intermediately-typed handler already satisfies post-condition 5, it is easy for the SMT solver to finish the proof needed to type the handler with the strong type req_handler.

SCIO* uses higher-order contracts not only during compilation but also during back-translation, the dual of compilation. For the secure compilation proofs in §6.4, we have to define back-translation from a target context with a weak interface to a source context that is still flag polymorphic, which involved designing our higher-order contracts mechanism so that it can handle flag-based effect polymorphism. For that we had to abstract away the enforcement of the checks from the higher-order contracts and introduced a new notion of *effectful checks* (see §4.2).

2.7 Enforcing the access control policy by reference monitoring

In our example, the assumption that we have to enforce using reference monitoring is post-condition 5 of the handler, stating that it can only open, read, and close its own files, and that only the web server is allowed to write during the execution (when the handler calls send). Our traces are informative so that we can distinguish between the events produced by the partial program (e.g., web server) and those produced by the context (e.g., handler). This is because every event of the trace contains a bit of information of type caller that can have the value Prog or Ctx. This bit allows us to enforce a stronger specification on the untrusted context than on the partial program—e.g., the handler *cannot* directly write to any file descriptor, but the web server can write to the clients.

We use the fact that the monitor enforces an access control policy on the context to give it a specification. However, using the policy directly to give specification would say only half of the

[4]We allow for error recovery, since it would be unacceptable for our web server to crash whenever a dynamic check fails. Our secure compilation theorem allows the information flow entailed by error recovery, as discussed in §6.4.

story because the policy characterizes only what the context can do—i.e., it would not specify what the partial program does in case the context calls (back) the partial program. Because of this, the SCIO* framework requires both an access control policy, denoted by Π of type policy, and its specification, denoted by Σ of type policy_spec.

```
type policy_spec = h:trace → caller → op:io_ops → arg:mio_sig.args op → Type0
type policy (Σ:policy_spec) = h:trace → op:io_ops → arg:mio_sig.args op → r:bool{r ⟹ Σ h Ctx op arg}
```

The access control policy is enforced *only* on the IO calls of the context, while its specification characterizes the IO calls of the context *and* the IO calls of the partial program during the execution of the context, which is reflected by the extra caller argument above. The refinement on the returned value of the policy makes sure that the policy implies the specification for the events of the context.

Here we give an example of an access control policy Π and its specification Σ that enforces the post-condition of the handler (5). The intuition is that the policy Π allows the handler only to open, read, and close its own files, while the specification Σ says that the trace produced by the handler contains the events allowed by the policy plus the writes done by the send callback.

```
val Σ : policy_spec
let Σ h caller op arg : Type0 =
match caller, op, arg with
| Ctx, Openfile, fnm →
fnm `in_folder` "/temp"
| Ctx, Read, fd → is_opened_by_Ctx fd h
| Ctx, Close, fd → is_opened_by_Ctx fd h
| Prog, Write, fd → ⊤
| _, _, _ → ⊥
```

```
val Π : policy Σ
let Π h op arg : bool =
match op, arg with
| Openfile, fnm → fnm `in_folder` "/temp"
| Read, fd → is_opened_by_Ctx fd h
| Close, fd → is_opened_by_Ctx fd h
| _, _ → false
```

We can encode such a policy as a post-condition by stating that each event that happened during the execution satisfies Σ. For that we use the following function enforced_locally to encode it as MIO a fl (λ_ → ⊤) (λh lt → enforced_locally Σ h lt), which we shorten to MIO a fl ⊤ Σ (from §2.3).

```
let rec enforced_locally (Σ : policy_spec) (h lt: trace) : Type0 =
  match lt with | [] → ⊤ | e :: t → let (| caller, op, arg, _ |) = destruct_event e in
                                    Σ h caller op arg ∧ enforced_locally Σ (e::h) t
```

The following three examples of adversarial request handlers all try to violate the access control policy: handler3 tries to open a file outside of the /temp folder, which is the only folder authorized by the contract; handler4 tries to write directly to the client, bypassing the send; handler5 tries to use IO operations outside of those authorized by opening a socket. All of them are prevented from executing by the reference monitor.

```
let adversarial_handler3 sec_io client req send = sec_io Openfile ("/etc/passwd",[O_RDWR],0x650)
let adversarial_handler4 sec_io client req send = sec_io Write (client,(Bytes.of_string "hello"))
let adversarial_handler5 sec_io client req send = sec_io Socket ()
```

These examples are parametric in the IO library (sec_io) as described in §2.4, which was hidden for simplicity in the previous examples. The type of the IO library (io_lib) is defined using a single dependent function that takes as arguments the operation (e.g., Read) and the arguments (e.g., file descriptor) and returns the result of the IO operation (e.g., buffer). The post-condition of the IO library guarantees what we mentioned earlier, that the reference monitor enforces the access control policy and if an IO operation is prevented from executing the trace does not change.

```
type io_lib (fl:erased tflag) (Σ:policy_spec) = (op : io_ops) → (arg : mio_sig.args op) →
  MIO (mio_sig.res op arg) fl ⊤ (ensures (λ h r lt → enforced_locally Σ h lt ∧
                                                    (match r with
                                                     | Inr Contract_failure → lt == []
                                                     | r' → lt == [convert_call_to_event Ctx op arg r'])))
```

To generate the secure IO library, we wrap the default IO library (denoted by call_io) into new functions that when invoked first enforce the access control policy Π, by using the enforce_policy function below. We enforce the access control policy on each operation by retrieving the monitor state using the get_mstate operation, and checking if the operation is allowed or not. For the secure IO library, the caller is always set to be the context (denoted by Ctx).

```
val enforce_policy : #Σ:policy_spec → Π:policy Σ → io_lib Σ AllOps
let enforce_policy Π op arg = if Π (get_mstate ()) op arg then call_io Ctx op arg else (Inr Contract_failure)
```

3 THE MIO MONADIC EFFECT

In this section we explain how we represent IO programs and their specifications using a new monadic effect that we call MIO (for *Monitored IO*). We begin by introducing a general construction for obtaining monadic effects and then explain how to set up each of the ingredients required for this construction: computations, specifications, and how these last two relate (§3.1). We finally show how IO operations are defined in MIO (§3.2).

3.1 The Dijkstra monad behind the MIO monadic effect

We seek to write IO computations as terms of a type MIO α fl pre post. The arguments to MIO can be summarized as follows: α is the return type of the computation, fl constrains which IO operations the computation can use (§2.4), pre is a pre-condition over the trace of past IO events (which we sometimes refer to as *history*) that must be satisfied to be able to call the computation, and post is a post-condition over the result and the trace produced by the current computation.

To define our MIO effect, we use a monad-like structure indexed by specifications known as a Dijkstra monad [Ahman et al. 2017; Maillard et al. 2019]. For a Dijkstra monad $\mathcal{D}$, a type $\mathcal{D}\ \alpha\ w$ classifies effectful computations returning values in α and specified by $w : W\ A$, where W is called a *specification monad*. We follow Maillard et al. [2019], and construct a Dijkstra monad for MIO by picking three ingredients: i) a computational monad, ii) a specification monad, and iii) a monad morphism from the former to the latter; obtaining thus an effect in which computations are indexed by specifications. This effect captures the return type (α) and specification (pre and post). The flag index (fl) described in §2.4 and §2.7 for controlling what operations the computation can access, is added by refining the effect with an extra index following Rastogi et al. [2021]. In the following paragraphs, we provide a concise summary of the ingredients above, and elaborate them in more detail in the subsequent subsections.

For the computational monad, we use a free monad that is parametric in the underlying primitive operations. Free monads are particularly advantageous when it comes to representing IO [Letan and Régis-Gianas 2020; Maillard et al. 2019; Swierstra and Altenkirch 2007], as its computations have an inherent tree structure. In these trees, each node corresponds to a call to an operation, encapsulating both the operation itself and its arguments, which act as *output* of the program. Furthermore, each operation node has a corresponding child node for each result, which act as *input* for the program. For simplicity, we define operations for file management and for socket communication, but these account only for one possible instantiation of the monad. Our approach can be extended to other IO operations as needed. Moreover, the free monad could be used to implement other effects such as exceptions, state, and non-determinism by extending the signature of effects [Bauer and Pretnar 2015], and we plan to integrate some of these effects in the future.

The MIO monadic effect uses a specification monad that captures the behavior of a computation as a predicate transformer, i.e., a function that given a post-condition, returns a pre-condition that is strong enough to guarantee the post-condition after execution of the computation. As explained in §1, in our setting a pre-condition is a property of the current trace, and a post-condition is a property of the result and the new trace. When we write specifications, we work directly with the

pre- and post-condition, and not with the predicate transformer. This is justified by the existence of a translation from pre- and post-condition pairs to predicate transformers [Maillard et al. 2019].

Finally, we need to define a monad morphism from the computational monad to the specification monad, so from a computation in the free monad to a predicate transformer. These steps allow us to build a Dijkstra monad for statically verifying IO programs with respect to specifications, but we also need a mechanism to support dynamic checks. For this we include a new silent operation called get_mstate that returns the reference monitor state, which allows us to write dynamic checks.

Dijkstra monads can be implemented in other proof oriented programming languages, not only in F⋆. For instance, Maillard et al. [2019] have also implemented Dijkstra monads in Coq. Nevertheless, F⋆ stands out due to its built-in support for SMT automation and the ease of defining new effects, making it a particularly convenient language for working with Dijkstra monads.

The computational monad. For the MIO effect, we chose a computational monad that is parametric in the underlying primitive operations by using a free monad [Bauer and Pretnar 2015] that can accept any signature of effects [Letan et al. 2021]. A signature is described by a type of operations together with the arguments' and result type for each operation:

```
type ops_sig (ops:Type) = { args : ops → Type ; res : (op:ops) → args op → Type }
```

We use the usual representation of the free monad, as trees whose nodes are operation calls, and whose leaves are the values returned by a computation.[5] Our Call constructor comes with an additional parameter of the variant type caller (§2.7) that we use to mark whether the computation making the call is the partial program (Prog) or the context (Ctx).

```
type free (ops:Type) (s:ops_sig ops) (a:Type) : Type =
| Call : caller → (op:ops) → (arg:s.args op) → cont:(s.res op arg → free ops s a) → free ops s a
| Return : a → free ops s a
```

The type constructor free forms a monad independently of the operations and signature, with free_return and free_bind combinators we define in the usual way [Bauer and Pretnar 2015].

For MIO, the signature we are interested in is given by a type of operations:

```
type mio_ops = | Openfile | Read | Write | Close | Socket | Setsockopt | Bind | SetNonblock | ... | GetMState
```

The first operations are natural IO operations (captured by a sub-type io_ops), while the last one is the operation that allows performing dynamic verification by retrieving the current trace. The three dots indicate the existence of more IO operations, such as those related to network communication that we use in §2, but which we do not write here for the sake of concision.

The signature we give enables all IO operations to return errors, as their result type is defined to be either a proper result or an error. For example, the Read operation has as argument type file_descr, and as return type either buffer err. The GetMState operation has argument type unit and result type the type of the monitor state. For simplicity, we assume here GetMState to return the current trace until §5, where we update it to return an abstract state. The computational monad is obtained by instantiating free with the type of the operations (mio_ops) and their signature (mio_sig):

```
type m a = free mio_ops mio_sig a
```

The specification monad. The role of the specification monad is to give a logical account of the semantics of the computation. We use a predicate transformer semantics for the specification of computations, in which predicates are properties written over a trace of past IO events, with traces represented simply as lists of events. We chose a specification monad that allows writing a pre-condition over the entire history of events, while the post-condition is written over the result of the computation and the events produced by it (we refer to it as local trace). These events are defined following the signature of the operations. For each IO operation, we have an event constructor capturing the operation name, caller, argument and result of the operation call:

[5] A special constructor of our free monad that is needed in F⋆ [Winterhalter et al. 2022] is omitted from our presentation.

```
type event = | EOpenfile : (c:caller) → a:mio_sig.args Openfile → (r:mio_sig.res Openfile a) → event | ...
```

For better SMT automation, we have opted to define events as a variant type, with each operation having its own constructor. This approach proved to be more effective in F⋆ compared to using dependent triples based on io_ops. Finally, GetMState does not have an event associated.

We define a type constructor w that captures transformers from post-conditions to pre-conditions:

```
type trace = list event
type w a = w_post:(lt:trace → r:a → Type0) → w_pre:(h:trace → Type0)
```

In this representation of specification, we interpret the history of events in pre-conditions as given in reverse chronological order. This is another factor contributing to successful SMT automation: Consider for instance the is_open predicate that checks whether some file descriptor fd is open according to the history. is_open proceeds by looking at events in the trace to see whether there is some EOpenfile _ (Inl fd) event that was not followed by an EClose fd _ event. When such a predicate is checked, most often the full history is not known, rather it is some abstract value h. Therefore, when one adds more events, the abstract history chunk is found at the end, last_event :: h, which is good because the SMT can reduce is_open fd (last_event :: h) to true if last_event is EOpenfile _ (Inl fd).

Predicate transformers can be naturally organized as continuation-like monads [Jensen 1978]. Indeed, as in the case of free, the type w comes equipped with a monadic structure given by combinators w_return and w_bind. In addition, a specification monad also needs to come equipped with an order between computations of the same type. For w, we define this order as follows:

```
let (⊑) wp1 wp2 = ∀h post. wp2 post h ⟹ wp1 post h
```

This order is a form of refinement [Swierstra and Baanen 2019] that compares specifications by precision. Combinators w_return and w_bind are monotone with respect to this order. To form an ordered monad we restrict to only those predicate transformers that are monotonic—i.e., which map stronger post-conditions to stronger pre-conditions—which we enforce by refining type w.

The monad morphism and the Dijkstra monad. As the third ingredient for constructing a Dijkstra monad, we need a monad morphism from the computational monad m to the specification monad w. We call such a morphism θ and it has type (#a:Type) → m a → w a. It is defined by recursion on the monadic computation, and uses the following function

```
let mio_wps (c:caller) (op:mio_ops) (arg:mio_sig.args op) : w (mio_sig.res op arg) = λpost h →
  if GetMState? op then post [] h else (∀ (r:mio_sig.res op arg). post [convert_call_to_event c op arg r] r)
```

that maps an operation and its arguments to a specification. For GetMState, this function captures the history, and passes it as the result to the post-condition (in §5 we change this definition to work with an abstract state). For IO operations, the post-condition receives a single-event trace corresponding to the IO operation performed, which is converted by the auxiliary function convert_call_to_event.

We proved that θ is indeed a monad morphism [Maillard et al. 2019, 2020], which means that it preserves the return and bind of mio and w. The representation of our Dijkstra monad is then:

```
type dm (a:Type) (wp:w a) = c:(m a){θ c ⊑ wp}
```

The idea is that a computation is indexed by wp if its image by θ is smaller than wp, which means that the specification computed by θ is refined by the one given in the index.

The layered effect. The final step for defining MIO is to add the flag that restricts the IO operations, which is needed to write *flag-polymorphic* context types (§2.4). We use a predicate satisfies : m α → tflag → bool that decides if the underlying computation uses only the operations allowed by the flag. The representation of the MIO monadic effect is then defined as follows:

```
type mio (a:Type) (fl:erased tflag) (wp:w a) = c:(dm a wp){c `satisfies` fl}
```

Given all the ingredients above, we use one of the F⋆ extension mechanisms [Rastogi et al. 2021] to define the MIO effect:

```
effect MIO (a:Type) (fl:erased tflag) (pre : trace → Type0) (post : trace → a → trace → Type0) = ...
```

By defining MIO, we can write F⋆ code in a direct and ML-like applicative syntax (Figure 2) that automatically elaborates into the monadic combinators. More importantly, F⋆ uses the specification monad to compute verification conditions and discharge them using the SMT solver.

3.2 The MIO effect operations

We have introduced the MIO effect on top a free monad that supports constructors such as Openfile, Read, Write, and Close. However, MIO computations do not need to know about these underlying constructors. Instead, we provide a function that wraps the IO operations and provide a meaningful specification derived from mio_wps:

```
let call_io (c : caller) (op : io_ops) (arg : mio_sig.args op) :
  MIO (mio_sig.res op arg) IOOps (requires (λ h → ⊤))
      (ensures (λ h (r:mio_sig.res op arg) lt → lt == [convert_call_to_event caller op arg r])) =
  MIO?.reflect (Call c op arg Return)
```

This specification asserts that the computation will have as a local trace a single event which corresponds to the performed operation. The flag used is IOOps, which indicates that the computation performs (only) IO operations. For writing code, we define synonyms for call_io for each IO operation, such as openfile, read and close. Similarly, we also define get_mstate, a wrapper for the operation GetMState, that takes no arguments and returns a trace (for now, we change this in §5). The flag for its effect is GetMStateOps. The specification ensures that the result of calling get_mstate is the history (r == h), and moreover, that the local trace is empty (no IO events happened, lt == []).

```
let get_mstate () : MIO trace GetMStateOps (requires (λ h → ⊤)) (ensures (λ h r lt → r == h ∧ lt == [])) =
  MIO?.reflect (Call Prog GetMState () Return)
```

4 HIGHER-ORDER CONTRACTS

We gave a brief presentation of how higher-order contracts work in §2.6, and now we give a more detailed picture of how the import and export functions are implemented. The two functions are designed to be used during the compilation of the partial program and during the back-translation of the context, which are explained later in §6, so the design was influenced by the need to work with flag-based effect polymorphism and by the need to define back-translation. As we said, import and export are two dual functions that convert between strongly-typed values and intermediately-typed values. Because the import and export functions are implemented in F⋆ and the conversion happens between F⋆ types directly, the correctness of the type conversion is verified by F⋆ typing.

4.1 Intermediate types

To begin, we define what we mean by strong types and intermediate types. We define *intermediate types* as a subset of F⋆ types, by using the trivial interm type class from Figure 3. The interm instances represent the types of the target language after the reference monitor was added to gave them a specification by enforcing the access control policy (§2.7). We provide instances for every base (unrefined) type, as well as pairs, sums, options, and non-dependent functions. The interm type class is indexed by an operations flag (fl, introduced in §2.4) and the specification of an access control policy (Σ). These indices ensure that higher-order functions are invariant over the flag fl and the policy Σ. This is realized by only defining instances that respect this property. We need intermediately-typed higher-order functions to be invariant over the flag and the policy so that they are aligned with the weakly-typed higher-order functions which are also invariant over those.

We do not give a definition for *strong* types. We consider strong types the types from which we can export from and we can import into. This definition is open because we implement import

```
class interm (ityp:Type) (fl:tflag) (Σ:policy_spec) = {}
instance interm_unit fl Σ : interm unit fl Σ = {}
instance interm_file_descr fl Σ : interm file_descr fl Σ = {}
...
instance interm_err fl Σ : interm err fl Σ = {}
instance interm_pair fl Σ t1 {| interm t1 fl Σ |} t2 {| interm t2 fl Σ |} : interm (t1 * t2) fl Σ = {}
instance interm_either fl Σ t1 {| interm t1 fl Σ |} t2 {| interm t2 fl Σ |} : interm (either t1 t2) fl Σ = {}
instance interm_arrow fl Σ t1 {| interm t1 fl Σ |} t2 {| interm t2 fl Σ |} : interm (t1 → MIO t2 fl ⊤ Σ) fl Σ = {}
```

Fig. 3. The type class representing intermediate types.

and export using type classes. We define instances for intermediate types, base refined types, pairs, sums, options, and functions with pre- and post-conditions.

4.2 Exportable and importable type classes

The import and export functions are fields of two type classes named importable_to and exportable_from. The classes are indexed by the *strong* type styp and they have as a field the *intermediate* type ityp. Function export is defined as taking a value of the strong type styp and returning a value of the intermediate type ityp, and function import is defined as taking a value of the intermediate type ityp and returning either a value of the strong type styp or an error in case that the dynamic check failed. The two type classes are additionally indexed by the operations flag fl and the specification Σ; these extra indices are required by the interm_ityp constraint on ityp to be an intermediate type.

```
class exportable_from (styp : Type) (fl : erased tflag) (Σ : policy_spec) (cks : checks) = {
  ityp : Type; interm_ityp : interm ityp fl Σ; export : eff_checks fl cks → styp → ityp; }
class importable_to (styp : Type) (fl : erased tflag) (Σ : policy_spec) (cks : checks) = {
  ityp : Type; interm_ityp : interm ityp fl Σ; import : ityp → eff_checks fl cks → either styp err; }
```

One special case is functions that are flag-based effect polymorphic. We want the wrapping of an effect polymorphic function to result also in an effect polymorphic function. This is necessary later in §6.4 when we define back-translation from a target context to source context that are both flag-based effect polymorphic. Our contracts dynamically enforce pre- and post-conditions about IO behavior, for which we have to use the effectful get_mstate operation—however, we *cannot* use it directly without losing the effect polymorphism. Our solution is to abstract away how the enforcement of the checks is done by parameterizing the import and export functions over, what we call, *effectful checks*. The effectful checks are indexed by the same operations flag fl as for the wrapped function, so they can be used inside the contract without affecting the effect polymorphism. Effectful checks are obtained by merging the get_mstate operation with the checks. The compilation framework automatically converts the checks into effectful checks and passes them when calling the import or the export functions. This approach forces us to index the type classes with the collection of checks that have to be enforced.

The cks collection contains dynamic checks to enforce the pre- and post-conditions of the strong type styp. However, it does not include dynamic checks to enforce refinement types because we do not need to use the get_mstate operation to enforce them. To learn about how we convert the refinement types into dynamic checks, we refer the reader to the work of Tanter and Tabareau [2015] because we use the mechanism they proposed, and most refinement types could anyway also be easily converted to pre- and the post-conditions.

4.3 Exporting and importing functions

The situation is most interesting for functions, where exporting and importing make use of each other. The basic idea is that in order to export a function f, we create a intermediately-typed function that imports its argument, calls f, and exports the result back to an intermediate type [Findler and Felleisen 2002]. Essentially, the composition export · f · import. Dually, an intermediately-typed

function g is imported roughly as import · g · export. However, as import can fail (it returns a sum type), we also require the codomain of each imported and exported function to "include" errors. Below is the instance for exporting simple functions, i.e., of type $t_1 \to$ MIO fl (either t_2 err) $\top$ Σ without other pre- and post-conditions:

```
instance exportable_smpl_arr (t1 t2:Type) (fl:erased tflag) (Σ:policy_spec) (cks:checks{EmptyNode? cks})
  {| d1 : importable_to t1 fl Σ (left cks) |} {| d2 : exportable_from t2 fl Σ (right cks) |}
: exportable_from (t1 → MIO (either t2 err) fl ⊤ Σ) Σ fl cks
= { ityp = d1.ityp → MIO (either d2.ityp err) fl ⊤ Σ; interm_ityp = solve;
    export = (λ cks (f:(t1 → MIO (either t2 err) fl ⊤ Σ)) (x:d1.ityp) →
      match x' ← d1.import x (left cks); f x' with (** do-notation **)
      | Inr err → Inr err | Inl y' → Inl (d2.export (right cks) y') ) }
```

The function is exported to the type $i_1 \to$ MIO fl (either $i_2$err) $\top$ Σ, where i_1,i_2 are the intermediate types corresponding to the strong types t_1,t_2 according to their instances. F* automatically checks that the type of the function is intermediate too (using the interm_arrow instance from Figure 3). Exporting produces a function that imports its argument, applies f, and exports the result with Inl. If either importing or the function itself fail (returning Inr), the error is returned instead.

We see here that the cks collection has the structure of a binary tree, defined using the inductive type checks. Since this function does not have a pre-/post-condition to enforce, the root node must be an EmptyNode, and its left and right children are used to import t_1 and export t_2. For a function with pre- and post-conditions, of type a → MIO b pre post, the tree takes the shape Node ck left right, where ck is a dynamic check testing whether the post-condition post indeed holds, and left and right are sub-trees for a and b respectively. For structured, non-arrow types such as tuples or sums, we also use an EmptyNode left right node, as there is no immediate pre- and post-condition to enforce, but there may be some deeper within the type. The Leaf node is used for base types when there are no checks to perform.

When importing or exporting a function with pre- and post-conditions, the cks collection becomes important. The cks collection must contain a boolean predicate for each pre- and post-condition (that needs enforcement). Hence, the instances of importable_to and exportable_from require a cks collection and alongside a proof that each check implies the actual (propositional) pre- or post-condition. Crucially, the dynamic checks have the same structure as the post-condition so that they can also distinguish between the history and the local trace of a computation, and they can also distinguish the events of the partial program from the events of the context. Since the types are higher-order, the trace of a computation can contain events generated by both the partial program and the context.Therefore, we give the following type to dynamic checks:

```
type ck_typ (t1 t2 : Type) = t1 → h:trace → t2 → lt:trace → bool
```

Given the type of the argument and return value of a function, a dynamic check is a boolean predicate over the value of the argument, the history before the call was made, the return value, and the local trace (i.e., the events generated during the activation of the function).

To enforce the dynamic checks, the compiler automatically generates the corresponding effectful checks with the same tree structure. The effectful checks are passed as an argument to the import and export functions. An effectful check guarantees that it enforces the corresponding dynamic check. An effectful check is done in two steps: a "setup" and an actual check. First, for the setup, we must mark the point in history where the function was initially called, in order to able to determine its local trace when it returns. This is accomplished by the effectful check because after the first call when it captures the initial trace, it returns a second function that enforces the dynamic check.

Enforcing the post-condition when importing. We are now ready to import a function with an intermediate type into one with a strong type that has a pre- and a post-condition. The

pre-condition does not need to be dynamically checked, because one can always strengthen the pre-condition of a function using subtyping, thus we only have to enforce the post-condition.

Here we show just a combinator that is used in an importable_to instance that focuses only on enforcing the post-condition. The combinator takes an intermediately-typed function f, a dynamic check ck, and its effectful counterpart eff_ck, and returns a strongly-typed variant of f. The combinator first "sets up" the effectful check, obtaining the do_ck, then it calls f, and then it enforces the post-condition by running do_ck, returning an error if the check fails.

```
let enforce_post (t1 t2 : Type) (fl:erased tflag) (Σ:policy_spec)
  (ck : ck_typ t1 (either t2 err)) (eff_ck : eff_ck_typ fl ck)
  (pre : t1 → trace → Type0) (post : t1 → trace → either t2 err → trace → Type0)
  (c1_post : squash (∀ x h r lt. pre x h ∧ enforced_locally Σ h lt ∧ ck x h r lt ⟹ post x h r lt))
  (c2_post : squash (∀ x h lt. pre x h ∧ enforced_locally Σ h lt ⟹ post x h (Inr Contract_failure) lt))
  (f : t1 → MIO (either t2 err) fl ⊤ Σ)
: (x:t1) → MIO (either t2 err) fl (pre x) (post x)
= λx → let do_ck = eff_ck x in
       let r : either t2 err = f x in
       if do_ck r then r else Inr Contract_failure
```

To import a function, two constraints have to hold between the pre- and post-condition, the dynamic check, and the access control policy, constraints that ensure that the post-condition can be soundly enforced. The first constraint, c1_post, ensures that the policy together with the dynamic check enforce the post-condition. This constraint allows us to return the result when the check succeeds by giving us that the post-condition holds. The second constraint, c2_post, ensures that the post-condition can be enforced even if the dynamic check fails as long as the policy was enforced. This constraint allows us to return Inr Contract_failure when the check fails after we called the function.

Enforcing the pre-condition when exporting. Exporting a strongly-typed function implies converting its pre-condition into a dynamic check and involves a mechanism similar to the one shown above for post-conditions. The main difference is that checking pre-conditions does not require a setup-check split, and can be done at once. To denote a dynamic check for a pre-condition, we reuse the ck_typ type above, using unit for the codomain and providing an empty local trace to the checks. We show below the enforce_pre combinator that is used in instances of exportable_from:

```
let enforce_pre t1 t2 fl Σ
  (ck : ck_typ t1 unit) (eff_ck : eff_ck_typ fl ck)
  (pre : t1 → trace → Type0) (post : t1 → trace → either t2 err → trace → Type0)
  (c_pre : squash (∀ x h. ck x h () [] ⟹ pre h))
  (c_post : squash (∀ h lt r. pre h ∧ post h r lt ⟹ enforced_locally Σ h lt))
  (f : (x:t1 → MIO (either t2 err) fl (pre x) (post x)))
: (x:t1 → MIO (either t2 err) fl ⊤ Σ)
= λ(x:t1) → if eff_ck x () then f x else Inr Contract_failure
```

To export a function, also two constraints have to hold. The first constraint, c_pre, ensures that the dynamic check enforces the pre-condition. This constraint gives us that the pre-condition holds after running the effectful check, allowing us to call the function. The second constraint, c_post, ensures that we can weaken the post-condition to the access control policy by subtyping.

Effectful checks are the key technical novelty to define back-translation between a target and a source context that are both flag-based effect polymorphic (§6.4).

5 DYNAMIC ENFORCEMENT OVER A MONITOR STATE

So far, we have described our specifications and dynamic checks over IO traces. While traces are intuitive and very expressive, making them a fine choice for specifications, they are inadequate for an implementation as their size may grow without bound. We would rather store only the information that is *needed* to implement the relevant dynamic checks. For example, if all dynamic checks are about whether file descriptors are open or not, the monitor could keep a set of currently open file

descriptors and efficiently probe this set instead of inspecting a trace. Importantly, independent of this runtime aspect we wish to still write specifications over traces, to retain full generality and a single *lingua franca*. SCIO* allows for such setups with full flexibility, allowing the programmer to choose exactly what the runtime monitor state should be and how the checks are performed, while still guaranteeing correctness over traces. We explain the mechanism in the rest of this section.

5.1 The monitor state

The entire SCIO* development is in fact parametrized over a type of *monitor state*, which represents the runtime information needed to implement the dynamic checks. Most of the definitions shown previously actually have an extra argument of type mstate. We elided them for simplicity of the previous sections, but will make them explicit here.

```
type mstate = { typ : Type; abstracts : typ → trace → Type0 }
```

The mstate.typ type must be instantiated by the programmer, who chooses it according to the dynamic checks that the whole program must perform. For instance, the type list file_descr would be a sensible choice if all one needs to check is whether file descriptors are open. The monitor state is exactly mstate.typ. Program traces are never materialized: they only appear in specifications.

The programmer must also relate the monitoring state to the trace of the program by defining a predicate abstracts. This relation specifies how the monitor state abstractly models the trace, and must be kept invariably true when applied to the current monitoring state and the current (ghost, immaterial) trace. For open file descriptors, one could use (λfds h → ∀fd. fd `mem` fds ⟺ is_open fd h), stating that a descriptor fd is in the list if and only if it is open according to the current trace h.

So in fact, the get_mstate operation does not return a trace, but a monitor state. Its post-condition also guarantees that the returned state abstracts the current trace (which is still accessible in specifications). The implementation of mio_wps is adjusted similarly.

```
let get_mstate (#mst:mstate) () :
  MIO mst.typ GetMStateOps (requires (λ h → ⊤)) (ensures (λ h s lt → s `mst.abstracts` h ∧ lt == [])) =
  MIO?.reflect (Call Prog GetMState () Return)
```

The recording of IO operations in the monitor state is abstracted away in SCIO*. The lower-level implementation of the state updates is explained in §7, where we discuss execution in OCaml.

5.2 Enforcing the access control policy and the checks over the monitor state

The access control policy and higher-order contracts must also work over monitor states. A policy, which must soundly enforce a higher-level policy_spec over traces, now takes a monitor state as argument, instead of a trace. The policy must ensure that, when it returns true, the policy specification is satisfied for *any* trace that is abstracted by the current monitor state.

```
type policy (mst:mstate) (Σ:policy_spec) = s0:mst.typ →
    op:io_ops → arg:io_sig.args op → r:bool{r ⟹ (∀ h. s0 `mst.abstracts` h ⟹ Σ h Ctx op arg)}
```

Coming back to the open files example, this means that a policy can only inspect the set of files to decide whether the operation should be allowed, and the answer must be right for any trace that would lead to the current open file set.

As for the dynamic checks (ck_typ), they are defined over an initial and final state of the function:

```
type ck_typ (mst:mstate) (t1 t2 : Type) = t1 → s0:mst.typ → t2 → s1:mst.typ → bool
```

Relating these states to the trace is done in the pre-conditions of the import and export functions from §4.3. One is the first constraint required when exporting a function (c_pre from 4.3) and the second one is the first constraint required when importing a function (c1_post from 4.3):

```
(c_pre : squash (∀ x s0 s1 h. s0 `mst.abstracts` h ∧ s1 `mst.abstracts` h ∧ ck x s0 () s1 ⟹ pre x h))
(c1_post : squash (∀ s0 s1 x h r lt . pre x h ∧ enforced_locally Σ h lt ∧
        s0 `mst.abstracts` h ∧ s1 `mst.abstracts` (rev lt @ h) ∧ ck x s0 r s1 ⟹ post x h r lt))
```

As explained in §4.2, we have to abstract away the enforcement of the checks by using effectful checks. They are represented by the type eff_ck_typ below. Enforcing a dynamic check ck is done in two steps: a "setup" and an actual check. The setup is done by calling the effectful check, which captures the current state s_0 and returns it together with the second part of the effectful check. The returned state is labeled with erased, thus it cannot be used in the computation: it is only there in order to specify the effectful check. The second part, of type eff_ck_typ_cont, is called to enforce the dynamic check ck and guarantees that the property holds: (b ⟺ ck x s_0 y s_1). This function also returns an erased state, again only for specification purposes. This gives us the ability to enforce the dynamic check on two different states—e.g., capture s_0 before calling a function and s_1 after the function returns—and this enables us to enforce post-conditions. Inside the enforce_pre and enforce_post combinators, the specifications of the effectful check that s_0 and s_1 abstract the histories, plus the constraints, make it very easy to prove that the enforcement is done correctly.

```
type eff_ck_typ_cont (mst:mstate) (fl:erased tflag) (t1 t2:Type) (ck:ck_typ mst t1 t2) (x:t1) (s0:mst.typ) =
  y:t2 → MIO (erased mst.typ * bool) mst fl ⊤
       (ensures (λ h1 (s1, b) lt → lt == [] ∧ s1 `mst.abstracts` h1 ∧ (b ⟺ ck x s0 y s1)))
type eff_ck_typ (mst:mstate) (fl:erased tflag) (t1 t2:Type) (ck:ck_typ mst t1 t2) =
  x:t1 → MIO (s0:erased mst.typ & eff_ck_typ_cont mst fl t1 t2 ck x s0) mst fl ⊤
            (ensures (λ h0 (| s0, _ |) lt → s0 `mst.abstracts` h0 ∧ lt == []))
```

Converting a pure dynamic check into an effectful check is straightforward:

```
let make_check_eff mst (t1 t2:Type) (ck:ck_typ mst t1 t2) : eff_ck_typ mst GetMStateOps ck = λ(x:t1) →
  let s0 = get_mstate () in
  let cont = (λ (y:t2) → let s1 = get_mstate () in (hide s1, ck x s0 y s1)) in (| hide s0, cont |)
```

5.3 The web server's monitor state

For our case study, we choose the following implementation of the monitor state: (a) a list of file descriptors that were opened by the context, (b) a single boolean flag that determines whether the current request was answered, and (c) a list of the file descriptors that were written to. We do not need to track *all* of the open files, but only those opened by the context. This state contains enough information to enforce the access control policy (5), the pre-condition of send (3), and the first part of the post-condition of the handler (4), and in abstracts we can see a conjunction of all of these.

```
let myst : mstate = {
  typ = { ctx_opened : list file_descr ; responded : bool ; written : list file_descr };
  abstracts = (λ s h → (∀ fd. fd `mem` s.ctx_opened ⟺ is_opened_by_Ctx fd h) ∧
    (responded ⟺ ¬(did_not_respond h)) ∧ (∀ fd. fd `mem` s.written ⟺ wrote_to fd h)); }
```

The access control policy is already defined in §2.7 and to use it with our chosen state we have to replace the is_opened_by_Ctx fd h with fd `mem`s.ctx_opened. Below are the dynamic checks required to import the handler, defined as a cks collection of type checks (§4.3). The first node is for the post-condition of the handler (4) and the next one is for the pre-condition of send (3).

```
let handler_cks : checks myst =
  Node (| file_descr, unit, (λ client s0 _ s1 → not (client `mem` s0.written) && client `mem` s1.written) |)
    (Node (| Bytes.bytes, unit, (λ res s0 _ _ → not (s0.responded) && valid_http_response res) |) Leaf Leaf)
    Leaf
```

When instantiating the importable_to type class using these checks, F⋆ automatically proves that three of the four constraints hold, asking for help only to prove that the first dynamic check implies the post-condition of the handler. Moreover, SCIO⋆ guarantees that no other check is necessary.

6 SCIO⋆: FORMALLY SECURE COMPILATION FRAMEWORK

We now put all the pieces together and define the SCIO⋆ secure compilation framework (i.e., languages, compiler, and linker in §6.1), give semantics to the source and target languages (§6.2),

```
type interface^S = {
  ctype : erased tflag → Type;
  Σ : policy_spec;
  Π : policy Σ;
  cks : checks;
  importable_ctype : fl:erased tflag → importable_to (ctype fl) fl Σ cks;
  ψ : post_cond; }

type interface^T = {
  ctype : erased tflag → policy_spec → Type;
  Σ : policy_spec;
  Π : policy Σ;
  weak_ctype : fl:erased tflag → interm (ctype fl Σ) fl Σ; }

let compile_interface (I^S:interface^S) : interface^T = { (** denoted by I^S↓ **)
  Σ = I^S.Σ;
  Π = I^S.Π;
  ctype = (λ fl _ → (I^S.importable_ctype fl).ityp);
  weak_ctype = (λ fl → (I^S.importable_ctype fl).interm_ityp); }

type prog^S I^S = fl:erased tflag → I^S.ctype (fl+IOOps) → unit → MIO int (fl+IOOps) ⊤ I^S.ψ
type ctx^T I^T = fl:erased tflag → #Σ':erased policy_spec → io_lib fl Σ' → I^T.ctype fl Σ'

type prog^T I^T = I^T.ctype AllOps I^T.Σ → unit → MIO int AllOps ⊤ ⊤
type whole^T = unit → MIO int AllOps ⊤ ⊤
let link^T #I^T (P:prog^T I^T) (C:ctx^T I^T) = P (C AllOps (enforce_policy call_io I^T.Π)) (** denoted by C[P] **)

let compile_prog #I^S (P:prog^S I^S) : prog^T (I^S↓ ) = (** denoted by P↓ **)
  λ(C : (I^S↓ ).ctype AllOps I^S.Σ) → P AllOps (import C (make_checks_eff I^S.cks))

type ctx^S I^S = fl:erased tflag → io_lib fl I^S.Σ → eff_checks I^S.cks → I^S.ctype fl
type whole^S = ψ : post_cond & (unit → MIO int AllOps ⊤ ψ)
let link^S #I^S (P:prog^S I^S) (C:ctx^S I^S) = (** denoted by C[P] **)
  (| I^S.ψ, P AllOps (C AllOps (enforce_policy call_io I^S.Π) (make_checks_eff I^S.cks)) |)

let back_translate_ctx #I^S (C:ctx^T I^S↓ ) : ctx^S I^S = λfl sec_io → import (C fl sec_io) (** denoted by C^T↑ **)
```

Fig. 4. Secure compilation framework. Idealized mathematical notation.

and describe our machine-checked proofs that SCIO* soundly enforces a global trace property (§6.3) and satisfies by construction a strong secure compilation criterion (§6.4). We present this in the setting where the partial program has initial control and the context is the library, then also describe the dual setting when the context has initial control and the program is the library (§6.5).

6.1 Secure compilation framework

We first use the definitions from the previous sections to formally define our source and target languages, compiler, and linker—to which we jointly refer as the SCIO* secure compilation framework. We define our languages using shallow embeddings, which is a common way to express DSLs in F* [Bhargavan et al. 2021b; Fromherz et al. 2021; Protzenko et al. 2017], and which also simplifies our proofs. We represent the partial program as a function (since in F* modules are not first-class objects and one cannot reason about them). We now describe the setting where the partial program has the initial control, thus we make the partial program get the context as argument (after the context is instantiated with the flag, etc)—the setting in which the context starts is outlined in §6.5.

The SCIO⋆ framework is listed in Figure 4 and explained step by step below. As explained before, the partial program and the context share a higher-order interface. The difference between the source and the target language is that in the source language, the partial program and the context share a *source* interface, while in the target language they share a *target* interface. The source interface (type interface^S) is a dependent record type that contains the type of the context (denoted by ctype), the access control policy Π and its specification Σ (both explained in §2.7), and the dynamic checks, cks, that are enforced by the higher-order contracts (explained in §4). The interface also contains a type class constraint importable_ctype ensuring that the guarantees the type of the context (ctype) provides to the program can be enforced using the policy Π and the dynamic checks cks. The importable_ctype constraint also ensures that ctype is flag-polymorphic.

The target interface (type interface^T) is another record type that contains the type of the context (ctype), but this time, the field weak_ctype requires ctype to be a weak type (explained in §2.4). The interface also includes the policy Π and its specification Σ, which are used during target linking.

The type of the partial source program (prog^S) and of the target context (ctx^T) can also be found in Figure 4. The partial source program is a computation in the MIO monadic effect that can call IO operations (fl+IOOps), but that, since it is statically verified, has no need to call the get_mstate operation, so it is otherwise parametric in the flag. The type of the target context uses ctype from the target interface I^T and the constraint weak_ctype from I^T makes sure that ctype is a weak type parametric in the flag. Thus the target context *cannot* directly call the IO operations or the operation get_mstate because of flag-based effect polymorphism (as explained in §2.4), and has to instead use the secure IO operations (of type io_lib fl Σ′, type explained in §2.7) to do any IO.

The compilation of a partial source program (compile_prog) is defined as explained in §2.6. The result of the compilation is a partial target program (prog^T) that expects a target context that was instantiated so that it satisfies specification I^T?Σ. The MIO computation of the partial target program (type prog^T) has no pre- and post-condition and is indexed by the flag AllOps to be able to run the dynamic checks that call the get_mstate operation.

The target linking (link^T) first instantiates the target context and then it applies the partial program to the instantiated context. The context is instantiated with the AllOps flag (since the context calls into the secure IO operations and wrapped callbacks that use get_mstate) and with the secure IO operations, thus obtaining an instantiated context that satisfies specification I^T?Σ (as also explained in §2.7). The result of the target linking is a whole program (whole^T), which is a MIO computation with no pre- and post-condition, that takes a unit and returns an int.

6.2 Trace-producing semantics

We define a trace-producing semantics to reason about whole programs. For that, we use trace properties as predicates over complete IO traces, which are traces of terminated executions together with the final result of the whole program (which is an int, like in UNIX).

```
type trace_property = trace * int → Type0
```

Whole programs are computations in the monadic effect MIO, so to reason about them we have to reveal their monadic representation using the reify construct of F⋆ [Grimm et al. 2018]. Thus, we define the following trace-producing semantics function beh, used in security theorems (§6.3):

```
let beh (wt:whole^T) : trace_property = beh_m (reify wt)
```

beh is defined for whole target programs, but it can also be used to define trace-producing semantics for whole source programs. For concision, we write beh for both target and source programs.

Having reify reveal the underlying representation of the computation, we define the semantics function beh_m over the computational monad m. In its definition, we use the already presented monad morphism θ (3.1), which gives a weakest-pre-condition semantics to m, and which we

adapt into a trace-producing semantics by using first a backward predicate transformer and then a pre-/post-condition transformer [Maillard et al. 2019]:

```
let beh_m (wt : m int) : trace_property = λ(lt,res) → ∀p. θ wt p [] ⟹ p lt res
```

Because we use the monad morphism when defining beh_m, we can lift the post-condition of a computation to a trace property—for example, for a computation w of type mio int AllOps $\top$ ψ (3.1), where ψ is a post-condition, we can state that beh_m(w) $\subseteq$ ψ, where tp $\subseteq$ post is defined as ∀lt r. tp (lt, r) ⟹ post [] r lt. This follows naturally from the representation of the Dijkstra monad (explained in 3.1) because the refinement in the type of the monad becomes an extrinsic property.

6.3 Soundness of hybrid enforcement with respect to global trace property

In our source language, one can verify that the behavior of the partial program together with the context satisfy a global trace property. One can do this by encoding the global trace property into the post-condition of who has the initial control, so for now the partial program (the dual setting is explained in §6.5). For example, we verified that the web server "responds to every request" by encoding this property in the post-condition of the web server, and this property characterizes both the behavior of the web sever and of the request handler. Since the web server has the initial control, its post-condition naturally becomes the post-condition of the whole program.

We prove in F* that the global trace property verified in the source also holds in the target because of the dynamic checks added by SCIO*. Thus, we show that the added higher-order contracts and the enforced access control policy soundly enforce the global trace property.

When the partial program has the initial control, its post-condition (denoted by $I^S.\psi$ in Figure 4) is the global trace property and we show that the property holds after compilation and linking the compiled partial program against any target context.

Theorem 6.1 (Soundness). $\forall I^S.\ \forall P : \mathsf{prog}^S\ I^S.\ \forall C^T : \mathsf{ctx}^T\ I^S{\downarrow}\ .\ \mathsf{beh}(C^T[P{\downarrow}]) \subseteq I^S\!\psi$

Proof sketch. We unfold the compilation and the target linking and obtain that

$C^T[P{\downarrow}]$ == P AllOps (import (C^T AllOps (enforce_policy call_io $I^T\!\Pi$)) (make_checks_eff I^Scks)).

Before unfolding the term $C^T[P{\downarrow}]$ has the weak type whole^T = unit → MIO int AllOps $\top$ $\top$, but the term after unfolding is an instantiation of the source program P, which has the stronger type unit → MIO int AllOps $\top$ $I^S\psi$. From the term after unfolding, we can lift the post-condition to a trace property (as explained in §6.2) and have that beh(P AllOps (import (C^T ...) ...)) $\subseteq$ $I^S\psi$, which implies that also beh($C^T[P{\downarrow}]$) $\subseteq$ $I^S\psi$. □

The proof follows so quickly because of the intrinsic specifications that are part of the types of the source partial program and of the imported context. Crucially, we lifted the post-condition of the partial program to a extrinsic property about the whole program. Less obvious is the role of the specification of the imported context in the proof: the behavior of the partial program is verified, including on how and if it calls the context, based on the specification of the context. Thus, it is enough to pass to the partial program a context with the correct specifications—i.e., with the correct type. The specifications are given to the target context by instantiating the context with the secure IO operations which enforce the access control policy, and by the dynamic checks enforced by the import function which is verified to be correct. We know that the access control policy and the checks give the correct specification to the context because of the type constraint (I^Simportable_ctype) between them. So the proof of soundness is done modularly by typing and also heavily benefits from SMT automation in F* [Maillard et al. 2019; Swamy et al. 2016].

6.4 Robust Relational Hyperproperty Preservation (RrHP)

We prove that SCIO* robustly preserves relational hyperproperties, which is the strongest secure compilation criterion of Abate et al. [2019], and in particular stronger than full abstraction, as proved by Abate et al. [2019] for a determinate setting with IO that closely matches ours. Relational hyperproperties [Abate et al. 2019] are a very broad class of security properties that includes trace properties (such as safety and liveness), hyperproperties [Clarkson and Schneider 2010] (such as noninterference), and also properties that relate the behaviors of multiple programs (such as trace equivalence). Robust Relational Hyperproperty Preservation (RrHP) states that for any relational hyperproperty that a collection of source programs robustly satisfies—i.e., the programs satisfy this relational hyperproperty when each of them is linked with the same arbitrary source context—then the compilation of these programs will also robustly satisfy the same relational hyperproperty with respect to an arbitrary target context. Intuitively, in order to achieve such a strong criterion, the various parts of the secure compilation framework have to work together to provide enough protection to the compiled programs so that a linked target context doesn't have more attack power than a source context would have against the original source programs.

One challenge in stating RrHP in our setting is that, as explained in §2, SCIO* allows the program and the context to recover from contract and monitoring errors. This is necessary for functionality, as it would be unacceptable for our web server to crash whenever a dynamic check fails. This does trade off some security though, because the results of these dynamic checks can potentially be used by the target context to mount some attacks that rely on the information they indirectly reveal about the program and its secrets. Our RrHP theorem allows this information flow that improves functionality because we align the attack capabilities of the source and the target contexts,[6] so that the recoverable errors caused by enforcement are possible in both contexts. We achieve this by also explicitly passing the effectful checks to the source context, which enables the source context to mount the same kind of attacks as the target context.

To state RrHP, we present the definitions of source contexts and source linking from Figure 4. The definition of source contexts (ctx^S) differs from the definition of target contexts (ctx^T) in two ways. First, the type of source contexts is indexed by a source interface (I^S) instead of a target interface (I^T). Second, a source context has an extra argument for the effectful checks that we pass to it in order to align the attack capabilities of the source and target contexts. We pass the effectful checks to the source context during source linking (see link^S in Figure 4), in contrast to what happens at the target level where the compiled partial program is the one that passes the effectful checks to the imported target context. Finally, the source context also has to be parametric in the flag because it should not have access to the get_mstate operation, since otherwise it would be able to distinguish between different source programs and their secrets just by looking at the trace, which would mean that source programs could robustly satisfy only trace properties, but not hyperproperties, making the RrHP theorem much weaker.

We are now ready to state RrHP and we use a property-free characterization that was proposed by Abate et al. [2019] and that is generally better tailored for proofs:

Theorem 6.2 (Robust Relational Hyperproperty Preservation (RrHP)).

$$\forall I^S.\ \forall C^T : \mathsf{ctx}^T\ I^S{\downarrow}\ .\ \exists C^S : \mathsf{ctx}^S\ I^S.\ \forall P : \mathsf{prog}^S\ I^S.\ \mathsf{beh}(C^T[P{\downarrow}]) = \mathsf{beh}(C^S[P])$$

Proof sketch. Looking at the quantifier structure, to prove this theorem one has to create a source context by defining a back-translation that only takes the target context as argument. In our case, we can define back-translation by partially applying import (see back_translate_ctx in

[6] Aligning the attack capabilities of the source and target contexts is a common necessity in secure compilation [Abate et al. 2019; Patrignani et al. 2019].

Figure 4), making it very similar to what compilation does to the target context. These definitions of compilation, back-translation, and source linking allow us to prove that compiling the program and linking it with the context is syntactically equal to back-translating the context and linking it with the program: $\forall I^S.\ \forall C^T{:}\mathsf{ctx}^T\ I^S\downarrow\ .\ \forall P{:}\mathsf{prog}^S\ I^S.\ \ C^T[P\downarrow] = C^T\uparrow[P]$. This syntactic inversion law is proved by just unfolding the definitions and makes the proof of the RrHP criterion immediate. □

While RrHP is the strongest secure compilation criterion of Abate et al. [2019], and such criteria are generally challenging to prove [Abate et al. 2019; Devriese et al. 2017; Jacobs et al. 2022; New et al. 2016; Patrignani et al. 2019], we have set things up so that it holds by construction, so our proof is easy. This simplicity is possible because (1) our languages are shallowly embedded in F*; (2) we use flag-based effect polymorphism to model the context; and (3) we design our higher-order contracts mechanism so that we can define both compilation and back-translation. These elements allow us to avoid a sophisticated logical relations argument [Abate et al. 2019; Devriese et al. 2017; New et al. 2016] by instead proving a syntactic inversion law relating not only compilation and back-translation, but also source and target linking. Target linking is important here, because it adds the checks that enforce the access control policy on the IO operations of the context.

Defining the back-translation function is often the most interesting part of the proof of such secure compilation criteria. Also in our setting it is non-trivial to define back-translation: we had to redesign the higher-order contracts mechanism (as explained in §4.2) so that back-translation results in source contexts that are flag-based effect polymorphic.

6.5 Dual Setting: the context has initial control

In previous subsections we formally defined the SCIO* framework in the setting when the partial program has the initial control and uses the context as a library. In this subsection, we show that SCIO* and its security theorems also work in the dual setting—i.e., when the context has initial control and uses the partial program as a library. In this presentation we focus on the main differences compared to the previous setting (Figure 4). At the level of the interfaces, we replace the type of the context (ctype) with a type for the partial program (ptype) and redefine the notions of partial program and context so that the context takes the partial program as argument:

```
type prog^S I^S = fl:erased tflag → I^S?ptype (fl+IOOps)
type ctx^T I^T = fl:erased tflag → Σ':erased _ → io_lib fl Σ' → I^T!ptype fl → unit → MIO int fl ⊤ Σ'
```

The type of the partial program has to be exportable, since we have to prepare the partial program by exporting it before passing it to the context. The definition of compilation and back-translation change so that instead of importing the context, the partial program is exported.

For this dual setting, we had to redefine the soundness theorem. When the partial program has the initial control, it is statically verified to satisfy its post-condition, thus the soundness theorem guarantees that the resulting target whole program also satisfies this post-condition. When the unverified context has the initial control, however, we only know that it is monitored, thus, the soundness theorem for this setting guarantees instead that the resulting target whole program satisfies the specification of the access control policy Σ.

Theorem 6.3 (Soundness-Dual). $\forall I^S.\ \forall P : \mathsf{prog}^S\ I^S.\ \forall C^T : \mathsf{ctx}^T\ I^S\downarrow\ .\ \mathsf{beh}(C^T[P\downarrow]) \subseteq I^S?\Sigma$

We proved this soundness theorem using the same strategy as for Theorem 6.1. We also proved the RrHP theorem in this dual setting using the same strategy as for Theorem 6.2, since the same syntactic inversion law also holds in this setting. All proofs have been machine-checked in F*. The artifact also contains instantiations of this dual setting with some examples (one presented in §8).

6.6 Syntactic representation of target contexts

In §6.1 we defined target contexts as flag-polymorphic functions taking Σ' and io_lib as arguments:

```
type ctx^T I^T = fl:erased tflag → #Σ':erased policy_spec → io_lib fl Σ' → I^T.ctype fl Σ'
```

These functions are good enough to represent general contexts that make calls to the IO operations provided by the io_lib argument. However, this definition of contexts might not be entirely transparent about the expressive capabilities of the target contexts and the fact that they are unverified. To make these more apparent, we also introduce a syntactic representation of target contexts. For this, we define a small deeply embedded language and a total translation function taking an arbitrary syntactic expression in this language and translating it to a function of type $\mathrm{ctx}^T\,I^T$ representing a target context. Our deeply embedded language is a simply-typed lambda calculus with primitive types (such as bytes, int, etc.), and we wrote the adversarial handlers of §2 also as syntactic expressions in this language, and then translated those expressions to obtain handlers that behave, when executed, as the original ones do (§7). We have reproved Theorems 6.1 and 6.2 also when the target context is a syntactic expression, in which case target linking and back-translation do the extra step of first translating this expression into a context with a weak interface.

7 RUNNING THE CASE STUDY IN OCAML

We run our main case study by extracting both the compiled web server and the handlers from our target language (a subset of F⋆) to OCaml using the standard extraction mechanism of F⋆ and test that they execute as intended when linked with wrapped up versions of the realistic IO library of OCaml, which reads and writes to files and network sockets. While securing and formally verifying this extra step going from our target language to OCaml is out of scope for this work, this experiment still offers empirical evidence that several attacks attempted by the adversarial request handlers written in our target language are blocked by the dynamic checks of either the higher-order contracts or the reference monitor, even at the OCaml level. Therefore, the interesting compilation step we secure and verify here works as expected also after further extraction to OCaml, and this should provide a good base for the more challenging task of building a larger secure compilation framework that protects verified programs against arbitrary contexts written in a safe subset of OCaml (§10). The web server was linked against the adversarial handlers from §2 and a non-adversarial handler that responds to HTTP requests from a real browser.

To run the web server, we implemented the recording of IO operations in the monitor state in F⋆ using a variant of the state effect that extracts to native OCaml references. This allowed us to verify that the individual updates of the monitor state are done correctly. Our implementation is parametric in an initial state init_st that abstracts the empty trace, and a verified upd_st function that, given a monitor state and an event, produces a new state that abstracts the new trace.

```
type init_st (mst:mstate) = s:mst.typ{s `mst.abstracts` []}
type upd_st mst = s0:mst.typ → e:event → s1:mst.typ{∀ h. s0 `mst.abstracts` h ⟹ s1 `mst.abstracts` e::h}
```

8 OTHER CLASSES OF EXAMPLES

In this section, we illustrate the applicability of SCIO⋆ by presenting three more examples: (1) the dual setting of §6.5 where the context now has the initial control; (2) a higher-order context that returns a callback; (3) different instantiations of the monitor state, including a stateless monitor.

1) Dual setting. Our logging IO library prints to the console all the IO requests and guarantees that no IO requests from the context can happen without being logged. In this example, the context has the initial control and it gets the partial program as argument, where the partial program is a logging function that writes to the console its arguments. To obtain our desired guarantees, we pick a specification that states that each IO operation done by the context is preceded by a logging

operation done by the program, and each logging operation must be followed by an IO operation done by the context. We encode this as the specification of the access control policy:

```
let Σ h caller op arg : policy_spec = match caller, op with
  | Ctx, _ → h ≠ [] ∧ hd h == EWrite Prog (stdout, to_string op) (Inl ())
  | Prog, Write → h == [] ∨ get_caller (hd h) == Ctx
  | _ → ⊥
```

Therefore, the context is forced to call the program before being able to do any IO operations. We instantiated the SCIO* framework with this example, thus, thanks to Theorem 6.3, the resulting target whole program satisfies the following specification: no IO request from the context can happen without being logged.

2) Archiving library. In this example, the partial program has the initial control and uses an untrusted higher-order function (of type zip below) to archive files. The untrusted function takes as first argument the file descriptor of the archive, writes the main header of the archive to it, and then returns another function (of type zip_file below) that the program has to call to add files into the archive. The specification of the archiving function guarantees that it only reads and writes to the files opened by the partial program.

```
type zip_file = fd:file_descr → MIO (either unit err) mymst (requires (λ h → is_open fd h))
                                                           (ensures (λ _ _ lt → only_reads_and_writes_to_fds_opened_by_prog lt))
type zip = afd:file_descr → MIO (either zip_file err) mymst (requires (λ h → is_open afd h))
                                                           (ensures (λ _ _ lt → only_reads_and_writes_to_fds_opened_by_prog lt))
```

This highlights that SCIO* fully supports higher-order contexts, including ones returning functions.

3) The monitor state. Our artifact contains multiple examples that show different instantiations of the monitor state, in addition to the one of the web server from §5.3. The archiving library from 2 has as state the entire trace of past IO events. The IO logging library example has as monitor state only the last event that happened. We also have an example where the monitor is stateless. These different examples showcase that the monitor state can be chosen flexibly.

9 RELATED WORK

Secure compilation. There are many approaches to secure compilation [Abate et al. 2019; Patrignani et al. 2019], but to our knowledge, the only ones to support secure compilation of *formally verified programs against unverified contexts* are those of Agten et al. [2015] and Strydonck et al. [2021]. They protect programs verified with separation logic against adversarial contexts using protected module architectures [Agten et al. 2012, 2015; Patrignani et al. 2015] or linear capabilities [Skorstengaard et al. 2021]. While they focus on stateful code and prove full abstraction, SCIO* focuses on code that can perform IO, and we establish RrHP with machine-checked proofs in F*.

Static verification of IO. There is a lot of work on statically verifying *whole* IO programs [Åman Pohjola et al. 2019; Férée et al. 2018; Guéneau et al. 2017; Jacobs et al. 2011; Letan and Régis-Gianas 2020; Malecha et al. 2011; Penninckx et al. 2015, 2019]. Interaction trees [Xia et al. 2020] were used to define a program logic using a monad morphism in the style of Dijkstra monads [Silver and Zdancewic 2021] to verify non-terminating impure computations in Coq. A case study verifies an HTTP Key-Value Server [Zhang et al. 2021] that is part of the verified operating system CertiKOS [Gu et al. 2016]. The web server is written in C and the trace properties are verified in Coq, requiring the manual application of tactics to prove verification goals. SCIO* simplifies this kind of use cases by taking advantage of SMT automation, yet extending the MIO monadic effect with non-termination is future work. Finally, all this line of related work focuses on how to verify whole programs, and does not address the problem of secure compilation.

Dependent interoperability. Strong interfaces in SCIO* contain refinement types and pre- and post-conditions that can depend on function arguments and results. Converting refinement

types into dynamic checks is inspired by Tanter and Tabareau [2015], who introduce a mechanism based on type classes. We extend this idea to convert pre- and post-conditions, and to go beyond pure functions, addressing new challenges. We see SCIO* as a first step towards achieving secure compilation from F* to OCaml (§10). The next steps could be build upon work on *dependent interoperability* [Dagand et al. 2018; Osera et al. 2012].

Higher-order contracts. Findler and Felleisen [2002] pioneered higher-order contracts, now a standard feature of Racket [Flatt and PLT, Chapter 8]. Several works have explored extensions to stateful contracts, e.g., Disney et al. [2011] propose temporal higher-order contracts, Scholliers et al. [2015] propose computational contracts, and Tov and Pucella [2010] study stateful contracts for affine types. Stateful contracts can be added at the boundary between the partial program and the context, and they can be used to implement an IO reference monitor. In particular, Moore et al. [2016] propose authorization contracts for implementing access control monitors for software components. The key novelty of SCIO* is to integrate statically-verified code with untrusted code in a tool that is itself verified. It would be interesting to explore whether soft contract verification [Nguyen et al. 2014] could be used in SCIO* to eliminate dynamic checks that can be verified statically.

Runtime verification. There is extensive prior work on how to monitor program executions either by using runtime verification and instrumentation. Java-MOP is a framework for Monitoring-Oriented Programming (MOP) that builds on Aspect-Oriented Programming (AOP) in Java [Chen and Roşu 2005; Chen et al. 2004; Jin et al. 2012]. They focus on synthesizing the monitor from formal specifications (e.g., in LTL) to instrument the whole program. Automatically synthesizing the dynamic checks is an open challenge in our setting. In this paper, we focus instead on highly expressive specifications, on the programmability and efficiency of the checks, and on obtaining formal security guarantees, by automatically verifying that the checks are always enough to guarantee the specifications in the strong interface.

Our work currently only deals with a single policy. Prior work on flow-based monitors—intensively studied in the AOP literature from both the points of view of expressiveness [Douence et al. 2005; Tanter 2008] and efficiency [Avgustinov et al. 2007; Bodden et al. 2007; Masuhara et al. 2003]—could be helpful to extend our work to support multiple, localized policies for specific contexts and linking points.

Gradual verification. Bader et al. [2018] and Wise et al. [2020] propose gradual program verification to easily combine dynamic and static verification in the same language at a very fine granularity, using variants of Hoare logic or separation logic with *imprecise* logical formulas. SCIO* is a hybrid verification and compilation framework with a coarser interoperability granularity (program vs. context), and targets the IO effect.

10 CONCLUSIONS AND FUTURE WORK

Securely compiling verified code and linking it with adversarial unverified code is a general open research problem for all proof-oriented programming languages (Coq, Isabelle/HOL, Dafny, etc). In this paper we make substantial progress towards solving this important problem by proposing a formally secure compilation framework for protecting verified IO programs against unverified code. In particular, we provide machine-checked proofs that our framework soundly enforces a global trace property and, moreover, satisfies a secure compilation criterion called RrHP, which is stronger than full abstraction.

We see this as an important first step towards building a larger formally secure compilation framework from F* to a safe subset of OCaml. Protecting compiled programs against arbitrary OCaml contexts is challenging, and would go beyond the state of the art in formally secure compilation, which has so far never been achieved for such realistic languages, but we believe that our current work can serve as a good base for that. The side effects of OCaml will need to be

exposed in the source language, which requires a significant extension of the framework, yet our free monad representation for MIO is general enough to allow expressing more of the side effects of OCaml such as non-termination, exceptions,[7] and state. The secure compilation proof techniques for shallowly embedded languages we designed in this paper will have to be combined with proof techniques for deeply embedded languages [Abate et al. 2019; Patrignani et al. 2019], and the security enforcement and proof will have to be extended to include F*'s extraction to OCaml, which is similar to that of Coq [Letouzey 2008]. The hope is to be able to reuse the recent correctness proof of Coq extraction [Sozeau et al. 2020, 2023] in a bigger secure compilation proof [Abate et al. 2018; El-Korashy et al. 2021], and our strategy from §6.6 for defining the back-translation.

While our contributions are developed in the context of F*, we believe that many ideas of the paper could be applied to solving the same general problem also in other proof-oriented programming languages, especially those based on dependent type theory. Dijkstra monads have, in particular, also been implemented in Coq [Maillard et al. 2019], so in principle a direct port to Coq of our whole SCIO* framework seems possible, just that the MIO Dijkstra monad will not be as usable in Coq without support for discharging the gathered verification conditions with an SMT solver, which is a big advantage we get from using F* instead. For Coq a more usable solution could probably be built more quickly by targeting an interactive verification framework for IO programs, such as Ynot [Malecha et al. 2011], FreeSpec [Letan and Régis-Gianas 2020], ITrees [Xia et al. 2020], or Iris [Penninckx et al. 2019]. While the precise details will vary based on the choice of verification framework, some general ideas that we expect to be portable to other frameworks are: (1) the internal representation of MIO as a free monad with an extra GetMState operation, which is directly compatible with how FreeSpec and ITrees represent computations, and which could provide a model to the axioms used by Ynot; (2) the formally verified combination of higher-order contracts and reference monitoring; (3) the secure compilation proof of RrHP from §6.4, which should stay simple even in these frameworks, since all the main ingredients seem portable; (4) the soundness proof from §6.3, although this will likely become more manual.

DATA AVAILABILITY STATEMENT

This paper comes with an artifact in F* that contains a formalization of the contributions above. The artifact contains the SCIO* framework, the mechanized proofs of sound enforcement of a global trace property and of RrHP, as well as a few examples. The artifact is available on Zenodo [Andrici et al. 2023b] and on Github [Andrici et al. 2023a].

ACKNOWLEDGMENTS

We thank the POPL 2024, ICFP 2023, PriSC 2023, and ICFP SRC 2020 referees for their helpful reviews. This work was in part supported by the European Research Council under ERC Starting Grant SECOMP (715753), by the German Federal Ministry of Education and Research BMBF (grant 16KISK038, project 6GEM), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972. E.R. was supported by the Estonian Research Council starting grant PSG749.

[7] Another interesting idea one could investigate is preventing the context from catching exceptions raised by our enforcement mechanism, so as to reduce the amount of information these errors can reveal to the context and thus to further strengthen our secure compilation theorem (§6.4).

REFERENCES

Dafny reference manual, 2023.

J. Åman Pohjola, H. Rostedt, and M. O. Myreen. Characteristic formulae for liveness properties of non-terminating CakeML programs. *ITP*. 2019.

C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. *CCS*. 2018.

C. Abate, R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. *CSF*. 2019.

P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. *CSF*. 2012.

P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. *POPL*. 2015.

D. Ahman, C. Hriţcu, K. Maillard, G. Martínez, G. Plotkin, J. Protzenko, A. Rastogi, and N. Swamy. Dijkstra monads for free. *POPL*. 2017.

A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver. CertiCoq: A verified compiler for Coq. In *3rd Workshop on Coq for Programming Languages (CoqPL)*, 2017.

J. Anderson. Computer security technology planning study. ESD-TR-73-51, US Air Force Electronic Systems Division (1973). Section 4.1.1 `http://csrc.nist.gov/publications/history/ande72.pdf`, 1973.

C.-C. Andrici, S. Ciobâcă, C. Hriţcu, G. Martínez, E. Rivas, E. Tanter, and T. Winterhalter. Artifact for the POPL 2024 paper 'Securing Verified IO Programs Against Unverified Code in F*'. GitHub, 2023a.

C.-C. Andrici, S. Ciobâcă, C. Hriţcu, G. Martínez, E. Rivas, E. Tanter, and T. Winterhalter. Artifact for the POPL 2024 paper 'Securing Verified IO Programs Against Unverified Code in F*'. Zenodo, 2023b.

A. W. Appel. Modular verification for computer security. *CSF*. 2016.

A. Arasu, T. Ramananandro, A. Rastogi, N. Swamy, A. Fromherz, K. Hietala, B. Parno, and R. Ramamurthy. FastVer2: A provably correct monitor for concurrent, key-value stores. *CPP*. 2023.

P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. *OOPSLA*. 2007.

J. Bader, J. Aldrich, and É. Tanter. Gradual program verification. *VMCAI*. 2018.

A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1): 108–123, 2015.

K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. An in-depth symbolic security analysis of the ACME standard. *CCS*. 2021a.

K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. DY*: A modular symbolic verification framework for executable cryptographic protocol code. *IEEE EuroS&P*. 2021b.

E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. *ECOOP*. 2007.

B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. T. V. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. *USENIX Security*. 2017.

J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA):126:1–126:30, 2020.

F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for Java. *TACAS*. 2005.

F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-based framework for software development and analysis. *FMSE*. 2004.

M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.

D. R. Cok and K. R. M. Leino. Specifying the boundary between unverified and verified code. In W. Ahrendt, B. Beckert, R. Bubel, and E. B. Johnsen, editors, *The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. 2022.

P. Dagand, N. Tabareau, and É. Tanter. Foundations of dependent interoperability. *J. Funct. Program.*, 28:e9, 2018.

A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Z. Béguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue. Implementing and proving the TLS 1.3 record layer. *IEEE S&P*. 2017.

D. Devriese, M. Patrignani, F. Piessens, and S. Keuchel. Modular, fully-abstract compilation by approximate back-translation. *Log. Methods Comput. Sci.*, 13(4), 2017.

T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. *ICFP*. 2011.

R. Douence, P. Fradet, and M. Südholt. Trace-based aspects. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

A. El-Korashy, S. Tsampas, M. Patrignani, D. Devriese, D. Garg, and F. Piessens. CapablePtrs: Securely compiling partial programs using the pointers-as-capabilities principle. *CSF*. 2021.

H. Férée, J. Å. Pohjola, R. Kumar, S. Owens, M. O. Myreen, and S. Ho. Program verification in the presence of I/O - semantics, verified library routines, and verified applications. *VSTTE*. 2018.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. *ICFP*. 2002.

M. Flatt and PLT. The Racket reference. https://docs.racket-lang.org/reference/.
A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy. A verified, efficient embedding of a verifiable assembly language. *PACMPL*, 3(POPL), 2019.
A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martínez, D. Merigoux, and T. Ramananandro. Steel: proof-oriented programming in a dependently typed concurrent separation logic. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.
N. Grimm, K. Maillard, C. Fournet, C. Hriţcu, M. Maffei, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, and S. Zanella-Béguelin. A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. *CPP*, 2018.
R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. *OSDI*. 2016.
A. Guéneau, M. O. Myreen, R. Kumar, and M. Norrish. Verified characteristic formulae for CakeML. *ESOP*. 2017.
A. Guéneau, J. Hostert, S. Spies, M. Sammler, L. Birkedal, and D. Dreyer. Melocoton: A program logic for verified interoperability between ocaml and c. *Proc. ACM Program. Lang.*, 7(OOPSLA2), 2023.
T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. Storage systems are distributed systems (so verify them that way!). *OSDI*, 2020.
S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. *IEEE S&P*. 2022.
B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. *NFM*. 2011.
K. Jacobs, D. Devriese, and A. Timany. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.*, 6(OOPSLA1):1–27, 2022.
K. Jensen. Connection between Dijkstra's predicate-transformers and denotational continuation-semantics. DAIMI Report Series 7.86, 1978.
D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient parametric runtime monitoring framework. *ICSE*. 2012.
G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. A. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an operating-system kernel. *CACM*, 53(6):107–115, 2010.
X. Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.
T. Letan and Y. Régis-Gianas. FreeSpec: specifying, verifying, and executing impure computations in Coq. *CPP*. 2020.
T. Letan, Y. Régis-Gianas, P. Chifflier, and G. Hiet. Modular verification of programs with effects and effects handlers. *Formal Aspects Comput.*, 33(1):127–150, 2021.
P. Letouzey. Coq extraction, an overview. In *LTA '08*. 2008.
J. Li, A. Lattuada, Y. Zhou, J. Cameron, J. Howell, B. Parno, and C. Hawblitzel. Linear types for large-scale systems verification. *OOPSLA*, 2022.
K. Maillard, D. Ahman, R. Atkey, G. Martínez, C. Hriţcu, E. Rivas, and E. Tanter. Dijkstra monads for all. *PACMPL*, 3(ICFP), 2019.
K. Maillard, C. Hriţcu, E. Rivas, and A. Van Muylder. The next 700 relational program logics. *PACMPL*, 4(POPL):4:1–4:33, 2020.
G. Malecha, G. Morrisett, and R. Wisnesky. Trace-based verification of imperative programs with I/O. *J. Symb. Comput.*, 46 (2):95–118, 2011.
G. Martínez, D. Ahman, V. Dumitrescu, N. Giannarakis, C. Hawblitzel, C. Hriţcu, M. Narasimhamurthy, Z. Paraskevopoulou, C. Pit-Claudel, J. Protzenko, T. Ramananandro, A. Rastogi, and N. Swamy. Meta-F*: Proof automation with SMT, tactics, and metaprograms. *ESOP*. 2019.
H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. *CC*. 2003.
S. Moore, C. Dimoulas, R. B. Findler, M. Flatt, and S. Chong. Extensible access control with authorization contracts. *OOPSLA*. 2016.
T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: From general purpose to a proof of information flow enforcement. *IEEE S&P*. 2013.
M. S. New, W. J. Bowman, and A. Ahmed. Fully abstract compilation via universal embedding. *ICFP*. 2016.
P. C. Nguyen, S. Tobin-Hochstadt, and D. V. Horn. Soft contract verification. *ICFP*. 2014.
P. Osera, V. Sjöberg, and S. Zdancewic. Dependent interoperability. *PLPV*. 2012.
Z. Paraskevopoulou, J. M. Li, and A. W. Appel. Compositional optimizations for CertiCoq. *PACMPL*, 5(ICFP):1–30, 2021.
M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *TOPLAS*, 37(2):6:1–6:50, 2015.
M. Patrignani, A. Ahmed, and D. Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 2019.
D. Patterson, N. Mushtak, A. Wagner, and A. Ahmed. Semantic soundness for language interoperability. *PLDI*. 2022.

W. Penninckx, B. Jacobs, and F. Piessens. Sound, modular and compositional verification of the input/output behavior of programs. *ESOP*. 2015.
W. Penninckx, A. Timany, and B. Jacobs. Abstract I/O specification. *CoRR*, abs/1901.10541, 2019.
J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP):17:1–17:29, 2017.
J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Z. Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. *IEEE S&P*. 2020.
T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. *USENIX Security*. 2019.
X. Rao, A. L. Georges, M. Legoupil, C. Watt, J. Pichon-Pharabod, P. Gardner, and L. Birkedal. Iris-wasm: Robust and modular verification of webassembly programs. *Proc. ACM Program. Lang.*, 7(PLDI):1096–1120, 2023.
A. Rastogi, G. Martínez, A. Fromherz, T. Ramananandro, and N. Swamy. Programming and proving with indexed effects, 2021.
M. Sammler, S. Spies, Y. Song, E. D'Osualdo, R. Krebbers, D. Garg, and D. Dreyer. DimSum: A decentralized approach to multi-language semantics and verification. *Proc. ACM Program. Lang.*, 7(POPL):775–805, 2023.
C. Scholliers, É. Tanter, and W. D. Meuter. Computational contracts. *Sci. Comput. Program.*, 98:360–375, 2015.
L. Silver and S. Zdancewic. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.*, 5(POPL):1–28, 2021.
L. Skorstengaard, D. Devriese, and L. Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *J. Funct. Program.*, 31:e9, 2021.
M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq Coq correct! verification of type checking and erasure for coq, in Coq. *Proc. ACM Program. Lang.*, 4(POPL):8:1–8:28, 2020.
M. Sozeau, Y. Forster, M. Lennon-Bertrand, J. B. Nielsen, N. Tabareau, and T. Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. working paper or preprint, 2023.
T. V. Strydonck, F. Piessens, and D. Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *J. Funct. Program.*, 31:e6, 2021.
N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. *POPL*. 2016.
W. Swierstra and T. Altenkirch. Beauty in the beast. *Haskell Workshop*. 2007.
W. Swierstra and T. Baanen. A predicate transformer semantics for effects (functional pearl). *PACMPL*, 3(ICFP):103:1–103:26, 2019.
É. Tanter. Expressive scoping of dynamically-deployed aspects. *AOSD*. 2008.
É. Tanter and N. Tabareau. Gradual certified programming in Coq. *DLS*. 2015.
J. A. Tov and R. Pucella. Stateful contracts for affine types. *ESOP*. 2010.
T. Winterhalter, C.-C. Andrici, C. Hriţcu, K. Maillard, G. Martínez, and E. Rivas. Partial Dijkstra Monads for all. TYPES, 2022.
J. Wise, J. Bader, C. Wong, J. Aldrich, É. Tanter, and J. Sunshine. Gradual verification of recursive heap data structures. *Proc. ACM Program. Lang.*, 4(OOPSLA):228:1–228:28, 2020.
L. Xia, Y. Zakowski, P. He, C. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020.
Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021.
H. Zhang, W. Honoré, N. Koh, Y. Li, Y. Li, L. Xia, L. Beringer, W. Mansky, B. C. Pierce, and S. Zdancewic. Verifying an HTTP key-value server with interaction trees and VST. *ITP*. 2021.
J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. *CCS*. 2017.

Received 2023-07-11; accepted 2023-11-07