Securely Compiling F* Programs With IO and Then Linking Them Against Weakly-Typed Interfaces

(Extended Abstract)

Cezar-Constantin Andrici[1] Cătălin Hriţcu[1] Théo Winterhalter[2]

[1] MPI-SP, Bochum, Germany [2] Inria Saclay, France

Abstract. *We propose a secure compilation chain for statically verified partial programs with IO. The source language is a subset of F* in which one can write and statically verify a partial IO program that interacts with its context via a strongly-typed higher-order interface, which includes refinement types as well as pre- and post-conditions that can talk about past IO events. The target language is a subset of the source in which the compiled program can be securely linked with a context via a weakly-typed interface, without refinement types or pre- and post-conditions. Compilation converts the logical assumptions the program makes about the context to runtime checks, while linking instruments the context by adding a reference monitor to soundly enforce a global safety property. In addition to soundness, we proved in F* that our secure compilation chain satisfies by construction Robust Relational Hyperproperty Preservation, which is the strongest secure compilation criterion of Abate et al. (CSF'19).*

In the proof-oriented language F* [13] one can write a program and statically verify that it satisfies a specification. The problem is that in such languages one needs to statically verify the *whole* program to guarantee the specification. This is often unrealistic, since in practice one uses third-party libraries that have a weakly-typed interface—i.e., without specifications—because they are written in the languages to which F* extracts (e.g., OCaml or C). Even if such a library with a weakly-typed interface was (re)written in F*, to be able to use it one would have to strengthen its interface either (1) by verifying the library, which if done statically takes away the simplicity of using it, since static verification in F* involves significant user interaction and expertise, or (2) by simply assuming the library respects the strongly-typed interface, which is unsound. In this work we propose to soundly strengthen the interface of the library by *dynamically* verifying it.

Our compilation chain starts with a partial source program with Input-Output (IO) (written in a subset of F*) that interacts with its context via a strongly-typed higher-order interface, which includes refinement types as well as pre- and post-conditions that can talk about past IO events [2]. For example, a post-condition could specify that the context should satisfy the safety property "it never opens the file `/etc/passwd`", or that the context returns an open file descriptor.

To dynamically verify that the context satisfies safety properties we use runtime verification [4, 9, 12]. We *instrument* the contex by adding a reference monitor that keeps as internal state a trace of each IO operation as it is happening during the execution. We add the monitor by taking advantage of the fact that the partial program and the context share the IO operations they can perform, but we give them operations with different implementations during compilation (for the program) and linking (for the context). The partial program uses an implementation that executes the operation and then updates the monitor's state, while the context uses an implementation that first checks if a global property π would be respected by the operation, and if so executes it and then it updates the monitor's state. For example, to enforce that the context does not open the file `/etc/passwd`, the global property π would be defined as "block all Openfile operations of `/etc/passwd`".

To enforce additional logical assumptions of the returned values of the context, we can take advantage of the trace that the monitor keeps as state and add runtime checks during compilation. For example, to verify that the context returns an open file descriptor, we can look at the trace and check if the file descriptor is the result of an Openfile operation and that there is no Close operation in the meantime.

Below we start by presenting how our linking in the target language enforces the safety property π on each IO operation by instrumenting the target context with a reference monitor. We then explain how our compiler converts the additional logical assumptions the program makes about the context using refinement types and pre- and post-conditions to runtime checks done at the (higher-order) boundary between the program and the context.

Refinements and pre-post conditions. The source program can make additional logical assumptions about the context beyond just π in the refinements and pre- and post-conditions. When making an additional assumption, one has to also hand-pick a sound dynamic approximation of it that can be enforced at runtime if linked against a target context. While one can make any assumption about the context, not all of them have sound dynamic approximations that are both *precise* and *efficiently enforceable*, thus it becomes a design decision on what additional assumptions are made and what dynamic approximations are picked.

Our compiler from the source to the target language consists only of one stage that converts the additional logical assumptions into runtime checks using a wrapping technique strongly inspired by higher-order contracts [8]. For this we define a way to export source values to the target, and dually a way to import target values into the source; where export and import are defined in terms of each other based on the types of the values, which is needed for supporting higher-order interfaces. For example, when exporting a value of refinement type, the refinement is simply erased. When exporting a function type, the pre-condition is converted into a runtime check, the arguments are imported, the post-condition is erased, and the returned value is exported. When importing a value to a refined type, a runtime check is added to make sure that the refinement is satisfied. When importing a function type, the arguments are exported, the returned values are imported, the pre-condition is added, and a runtime check is performed for the additional logical assumption from the post-condition. These runtime checks are added only at the interface between the partial-program and the context.

Target language. Let P^T_π be a partial target program and C^T a target context, where we know that P^T_π is expecting a context that satisfies monitorable safety property π (a special kind of trace property) for its statical guarantees to hold, but where we do not know whether C^T satisfies it. Our target linking instruments C^T to add a monitor that dynamically enforces property π, thus C^T becoming C^T_π.

The partial program P and the context C are interacting computations that can perform IO. We represent IO computations using a free monad variant indexed by specifications that can encode trace properties [2]. The actions of our free monad include IO operations such as opening files, reading from and writing to file descriptors, and closing file descriptors. We denote the free monad indexed by specification with IIO α *fl pre post*, where IIO stands for instrumented IO, where α is the return type, *fl* index is representing which IO actions the computation can contain, *pre* is a precondition over the history of events 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 model that the target context can be linked with any IO implementation of the actions, we make the target context *flag polymorphic* and *π-polymorphic*. In a first-order setting, the type of the target context C^T is:
$\mathsf{fl{:}erased}\ \_ \to \pi{:}\mathsf{erased}\ \_ \to \mathsf{acts}\ \pi\ \mathsf{fl} \to \alpha \to \mathsf{IIO}\ \beta\ \mathsf{fl}\ \top\ \pi$ where fl is the flag (labeled with erased making it unusable in the computation), acts fl π is the signature of the IO actions in IIO that use only actions under flag fl and that satisfy π, and α and β are the argument and return type of the context. Because C^T is *flag polymorphic*, it can not use directly the IO actions and it has to use the implementation passed as argument. Because C^T is *π-polymorphic*, it can also accept implementations with any specification, and by instantiating it with a π and with a implementation of the IO actions that satisfy π, C^T also will satisfy π.

In a first-order setting, we can take the complete type of P^T_π as $(\alpha \to \mathsf{IIO}\ \beta\ \mathsf{AllActions}\ \top\ \pi) \to \mathsf{IIO}\ \mathbb{Z}\ \mathsf{AllActions}\ \top\ \top$. Then, target language linking is defined as a function application $C^T[P^T_\pi] = P^T_\pi\ C^T_\pi$,

where $C^T_\pi = C^T\ \mathsf{AllActions}\ \pi\ (\mathsf{instrument\ io\_acts}\ \pi)$.

We can in fact generate the IO actions passed to the context, because we can implement instrument above generically in the IIO monad. The IIO monad has an extra action GetTrace that returns the IO trace until now — this enables checking the safety property π dynamically before each IO call. GetTrace is a type of reflection and is not part of io_acts. We have set things up so that the source partial program and context and the target context *cannot* call GetTrace directly.

Security criteria. We proved in F* two security criteria for our compilation chain. Using the notations of the previous sections, we define compilation of the partial program as follows:

$$P^S_\pi \downarrow = \lambda\ C^T_\pi \to (P^S_\pi\ (\mathsf{import}\ C^T_\pi)) <: \mathsf{IIO}\ \mathbb{Z}\ \mathsf{AllActions}\ \top\ \top$$

Linking produces a whole program, about which we reason using trace-producing semantics. We denote that a whole program respects a safety property ψ with $\mathsf{Beh}(C[P]) \subseteq \psi$.

1. Soundness. We first show that the compiled source program linked with the target context respects the safety property π.

$$\forall \pi\ P^S_\pi\ C^T.\ \mathsf{Beh}(C^T[P^S_\pi \downarrow]) \subseteq \pi$$

Proof sketch. Linking produces an IIO π computation, thus soundness is ensured by F* typing, which relies on the soundness of the checks our reference monitor does. □

2. Robust Relational Hyperproperty Preservation (RrHP). We show about the compilation chain that it robustly preserves relational hyperproperties using the criterion RrHP, which is the strongest secure compilation criterion of Abate et al. [1].

$$\forall C^T.\ \exists C^S_\pi.\ \forall \pi\ P^S_\pi.\ \mathsf{Beh}(C^T[P^S_\pi \downarrow]) = \mathsf{Beh}(C^S_\pi[P^S_\pi])$$

Proof sketch. To prove such a criterion, one has to create a source context only from the target context by using backtranslation. In our case, we can define backtranslation like this:

$$C^T \uparrow = \lambda\ \mathsf{fl}\ \pi\ \mathsf{acts} \to \mathsf{import}\ (C^T\ \mathsf{fl}\ \pi\ \mathsf{acts})$$

Our compiler and linker are designed so that we can prove the following syntactic equality (by unfolding the definitions) which makes the proof of the criterion immediate:

$$\forall C^T \pi\ P^S_\pi.\ C^T[P^S_\pi \downarrow] = C^T \uparrow [P^S_\pi]$$

□

This secure compilation proof is basically *by construction* and orders of magnitude simpler than most other proofs in this space. This simplicity is possible since (1) our languages are shallowly embedded in F* and (2) we get backtranslation for free from the way we compile higher-order functions (which is, as mentioned above, reminiscent of higher-order contracts [8]).

Artifact. This extended abstract comes with an artifact in F⋆ that contains a formalization of the ideas above.[1] IO computations are implemented using the Dijkstra Monad [10, 14] of Andrici et al. [2]. The artifact contains the compilation chain and mechanized proofs of soundness and RrHP.

Future work. Our long term goal is to have a realistic secure compilation chain from F⋆ to a safe subset of OCaml. For this we need to extend our IIO Dijkstra Monad, which now only supports the IO effect, with other OCaml effects such as non-termination, exceptions, and state. As a case study, we are working on a simple web server that supports third-party plugins written in the target language.

Related work. Chen et al. [5] presented a framework to enable Monitoring Oriented Programming (MOP) for software development and analysis that builds on the Aspect Oriented Programming (AOP). They also present an environment [6] that implements their framework that enables MOP for Java. In their framework, "monitors are automatically synthesized from formal specifications and integrated at appropriate places in the program". It seems MOP can be used to solve the same problem as us, but our work differs from theirs in one major way. The MOP depends on the powerful Java Virtual Machine with AOP enabled; AOP is well developed in Java, but even though some work to bring this paradigm to different languages exists, it does not seem to be as well developed, therefore MOP, for now, is only possible in Java. Our proposal does not depend on AOP, and in fact, it can work with most languages, therefore our work is more general.

Bader et al. [3] and Wise et al. [15] propose gradual program verification to easily combine dynamic and static verification in the same language. The main difference is that our work tries to give a model that combines dynamic and static verification in a source and a target language.

Interoperability between trusted and untrusted code was also studied by Sammler et al. [11], by showing the benefits of low-level sandboxing. This method relies on affine types and works great with the import/export mechanism used generally in gradual typing. They have a similar notion of exposing a wrapped version of the operation to the untrusted side, that has runtime checks. But they discuss only robust safety related to the memory model and they do not discuss trace properties.

Dagand et al. [7] propose a dependent interoperability framework that has a mechanism to export dependently-typed programs to simply-typed languages. Their focus is on type-safety between the languages, and they do not discuss about the case in which the dependent-types are used to reason about the behavior of the source program by using traces. We on the other hand, start from a source program typed to satisfy trace properties and take care that the behavior is preserved.

Acknowledgments. We thank Ştefan Ciobâcă, Guido Martínez, Exequiel Rivas, and Éric Tanter for the insightful discussions about this work. We also thank the anonymous reviewers at PriSC'23 for their helpful feedback. 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.

References

[1] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 256–271. https://doi.org/10.1109/CSF.2019.00025

[2] Cezar-Constantin Andrici, Théo Winterhalter, and Cătălin Hriţcu. 2022. Verifying non-terminating programs with IO in F*. HOPE.

[3] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science)*, Isil Dillig and Jens Palsberg (Eds.), Vol. 10747. Springer, 25–46. https://doi.org/10.1007/978-3-319-73721-8_2

[4] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, Ezio Bartocci and Yliès Falcone (Eds.). Lecture Notes in Computer Science, Vol. 10457. Springer, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[5] Feng Chen, Marcelo D'Amorim, and Grigore Roşu. 2004. A Formal Monitoring-Based Framework for Software Development and Analysis. In *Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 357–372. https://doi.org/10.1007/978-3-540-30482-1_31

[6] Feng Chen and Grigore Roşu. 2005. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 546–550. https://doi.org/10.1007/978-3-540-31980-1_36

[7] Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of dependent interoperability. *J. Funct. Program.* 28 (2018), e9. https://doi.org/10.1017/S0956796818000011

[8] Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 48–59. https://doi.org/10.1145/581478.581484

[9] Leslie Lamport and Fred B. Schneider. 1984. Formal Foundation for Specification and Verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course, April 3-12, 1984 and April 16-25, 1985, Munich, Germany (Lecture Notes in Computer Science)*, Mack W. Alford, Jean-Pierre Ansart, Günter Hommel, Leslie Lamport, Barbara Liskov, Geoff P. Mullery, and Fred B. Schneider (Eds.), Vol. 190. Springer, 203–285. https://doi.org/10.1007/3-540-15216-4_15

[10] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for

[1] http://github.com/andricicezar/fstar-io/tree/prisc-submission

all. *Proc. ACM Program. Lang.* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

[11] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

[12] Fred B. Schneider. 2000. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (2000), 30–50. https://doi.org/10.1145/353323.353382

[13] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

[14] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '13)*. 387–398. https://www.microsoft.com/en-us/research/publication/verifying-higher-order-programs-with-the-dijkstra-monad/

[15] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 228:1–228:28. https://doi.org/10.1145/3428296