# KEM-IND-CCA-Preserving Compilation of Jasmin's ML-KEM


Santiago Arranz-Olmos
MPI-SP
Bochum, Germany
santiago.arranz-olmos@mpi-sp.org

Gilles Barthe
MPI-SP
Bochum, Germany
IMDEA Software Institute
Madrid, Spain
gilles.barthe@mpi-sp.org

Lionel Blatter
MPI-SP
Bochum, Germany
lionel.blatter@mpi-sp.org

Benjamin Grégoire
Inria
Sophia Antipolis, France
benjamin.gregoire@inria.fr

Vincent Laporte
Inria
Nancy, France
Université de Lorraine
Nancy, France
Vincent.Laporte@inria.fr

Paolo Torrini
Inria
Sophia Antipolis, France
paolo.torrini@inria.fr





## Abstract

High-assurance cryptography provides strong guarantees that source implementations are functionally correct and provably secure. In this paper, we demonstrate that the Jasmin compiler preserves functional correctness and KEM-IND-CCA security (which were established in prior work) of a highly optimized Jasmin implementation of ML-KEM used in the popular messenger Signal. Our proof of preservation is fully mechanized in the Rocq prover and is based on three general contributions: (1) A general framework for modeling game-based security and for reasoning about preservation of game-based security under compilation. (2) A new, interaction-trees-based semantics of Jasmin and assembly programs. Our new semantics supports features required by ML-KEM, such as probabilistic computations and rejection sampling routines. (3) A new relational Hoare logic for interaction trees, which we use to prove correctness of the Jasmin compiler under our new semantics.



## CCS Concepts

• **Security and privacy → Logic and verification**.

## Keywords

Compiler verification, game-based security, Jasmin, ML-KEM, Rocq


## 1 Introduction

Formally verified cryptography [16] provides strong guarantees that cryptographic schemes are provably secure and that implementations are functionally correct with respect to these schemes. Moreover, these guarantees can be composed to yield stronger assurance that implementations are both correct and secure. However, a final and significant step remains: proving that correctness and security guarantees carry to assembly code. This final step eliminates relying on compilers, which may contain bugs or introduce security concerns [39, 81].

This paper reports on an experiment to carry out this step for a concrete case: a highly optimized implementation of ML-KEM [45]. ML-KEM is the newly adopted FIPS 203 post-quantum key encapsulation mechanism [62]. ML-KEM is being actively deployed as part of the post-quantum transition, which is expected to be completed around 2035 [60]. For example, Google recently announced they aim to complete their transition by 2029 [2]. The implementation we target is written in Jasmin, a programming language for high-assurance cryptography [7, 9], and has been formally verified for correctness and indistinguishability against chosen ciphertext attacks (KEM-IND-CCA) [6, 10]—both proofs in the EasyCrypt proof assistant [24, 25]. As of 2026, the Jasmin implementation is used in the popular secure messaging application Signal [5].

Unfortunately, the current infrastructure of Jasmin does not suffice for carrying functional correctness and KEM-IND-CCA security from source Jasmin programs, such as ML-KEM, to generated assembly. The foremost reason is that ML-KEM performs probabilistic computations and in particular uses rejection sampling (a.k.a. repeat until success). Hence, executions of ML-KEM are potentially unbounded—in probabilistic programming jargon, ML-KEM satisfies a probabilistic notion of termination known as *almost sure termination*. However, the original semantics of Jasmin programs is limited to deterministic and terminating programs. To overcome these issues, we equip both Jasmin and assembly programs with a new denotational semantics that models probabilistic behavior, and we prove compiler correctness for these new semantics.

The second reason is that game-based security notions such as KEM-IND-CCA are modeled as adversarial, probabilistic (hyper)properties that fall outside the scope typically considered in compiler correctness and secure compilation [19], and in particular in the context of the Jasmin compiler. We address this limitation by defining a general framework to define game-based security and by lifting the compiler correctness result to show its preservation.

We address both challenges using interaction trees (ITrees) [80, 83], a semantic framework well-suited to capture game-based security. Intuitively, the reason is that *interactions* with an environment—which are a central notion in ITrees, realized as "events" and discussed later—support all relevant features of game-based security (queries, randomness, nontermination, statefulness, etc.).

*Contributions.* This paper makes the following contributions.

**(C1)** We define a general notion of security game, based on oracle systems [23, 33, 78], that captures common game-based security notions (e.g., indistinguishability and unforgeability) and provides a unified foundation for reasoning about source and target programs.

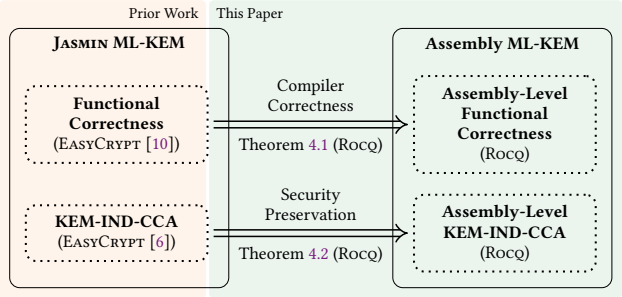


**Figure 1: Prior ML-KEM verification efforts and this work.**

**(C2)** We equip Jasmin and assembly programs with a denotational, probabilistic semantics, and we prove that the Jasmin compiler is correct with respect to this semantics.

**(C3)** We instantiate game-based security notions to Jasmin and assembly programs and show that compiler correctness implies preservation of all game-based security properties (Theorem 4.5). As a consequence, we get preservation of KEM-IND-CCA for the Jasmin vectorized implementation of ML-KEM, as shown in Figure 1 (prior work established KEM-IND-CCA security and functional correctness at the source level, our work defines these properties at assembly level and bridges the gap).

**(C4)** We develop a variant of relational Hoare logic for interaction trees. The logic, which is of independent interest and tailored for compiler correctness proofs, yields compact proofs.

*Artifact.* The Rocq mechanization of our framework and the version of Jasmin with its new proofs are open-source [13].

*Trusted Computing Base (TCB).* The TCB for our work comprises Jasmin's preprocessor and printer; its formalization of x86-64; and, naturally, the Rocq kernel and OCaml compiler.

*Future Work.* This paper specifically focuses on compiler preservation of game-based security. We consider the classic setting of game-based security, in which adversaries interact with cryptographic schemes through their specified interfaces, but cannot observe their side-channel leakage and cannot interfere with their execution, e.g., by controlling the cache or the branch predictors. While such stronger adversaries are out of scope of our work, the contributions of this paper provide a foundation upon which side-channel security guarantees can be built.

In follow-up work, we plan to extend the compiler security guarantees to stronger, system-level adversaries (e.g., that observe or carry out Spectre attacks). The main steps toward this goal are: (1) formalizing a (speculative) constant-time type system [36] *but at assembly level*; (2) proving that said type system enforces noninterference *in a system-level model of execution* [20]; (3) connecting the guarantees of the type system with the compiler's. Further details are discussed in the conclusion.

*Outline.* Section 2 introduces our framework for game-based security and reductions using ITrees. Throughout the paper, we use KEM-IND-CCA as an illustrative example, discussing the generalization to game-based security notions afterward. Section 3 defines the ITree semantics of the Jasmin language and the notion of security for Jasmin programs. Section 4 then states compiler correctness and leverages it to prove that Jasmin preserves game-based security. Section 5 presents the key proof technique used to prove compiler correctness: relational Hoare logic for ITrees. Lastly, Section 6 discusses the verification of the Jasmin compiler, illustrating each key idea by discussing a concrete compiler pass.

## 2 Game-Based Security with Interaction Trees

This section introduces our formalization of game-based security and reductions. For concreteness, we first focus on key encapsulation mechanisms (KEMs) and extend our methodology to arbitrary oracle systems afterward. Our approach is based on interaction trees [80, 83], a powerful semantic framework for modeling and reasoning about effectful computations. In order to keep our exposition accessible to a general audience, we introduce instances and features of interaction trees as needed. A more technical introduction is provided in the following sections and in Section A.

### 2.1 Key Encapsulation Mechanisms

Cryptographic schemes are modeled as sets of probabilistic algorithms. For instance, a KEM is given by three algorithms for key generation, encapsulation and decapsulation. Informally, key generation generates a key pair consisting of a public key and a secret key. Encapsulation takes as input a public key and generates a message and its ciphertext. Finally, decapsulation takes as input a secret key and a ciphertext and returns a message. We let $\mathcal{PK}$, $\mathcal{SK}$, $\mathcal{M}$ and $C$ denote the types of public keys, secret keys, messages, and ciphertexts, respectively. Our modeling of KEMs uses ITrees of type itree Rnd R, which represent probabilistic computations (Rnd corresponds to random sampling) that output results in R. Thus, KEMs are triples of the form:

$$\begin{aligned} \text{GenKey} &: () \to \text{itree Rnd } (\mathcal{PK} \times \mathcal{SK}), \\ \text{Encap} &: \mathcal{PK} \to \text{itree Rnd } (\mathcal{M} \times C), \\ \text{Decap} &: \mathcal{SK} \times C \to \text{itree Rnd } \mathcal{M}. \end{aligned}$$

That is, GenKey is a computation that samples randomness and returns a key pair. Similarly, decapsulation takes a secret and a ciphertext and returns a message; note that our model corresponds to KEMs with implicit rejection, in which Decap never fails. Modeling explicit rejection entails changing the return type of Decap to itree Rnd $(\mathcal{M} \cup \{\bot\})$, to indicate that computations may fail.

Critically, ITrees do not constrain the termination behavior of probabilistic computations; in particular, it is possible to model rejection sampling (or repeat until success), which is at the core of ML-KEM. The ability to model rejection sampling is a main challenge for our development.

### 2.2 Chosen Ciphertext Attack Security

The standard notion of security for a KEM is that an attacker cannot distinguish between a real message and a random one, given ciphertexts: this is called indistinguishability against chosen ciphertext attacks. The notion is defined using a probabilistic game between a challenger $\mathscr{C}$ with access to a KEM and an adversary $\mathscr{A}$. We model adversaries using interaction trees of the form itree (Dec +′ Rnd) R, which informally represent probabilistic computations that can perform decapsulation queries Dec that are answered with the KEM's

$$
\begin{aligned}
\text{KEM-IND-CCA}_{\mathscr{C},\mathscr{A}} \triangleq\ & pk, sk \leftarrow \text{GenKey}() ; \\
& s \leftarrow \langle \mathscr{A}_1(pk) \bowtie \text{Decap}(sk) \rangle ; \\
& m_0, c \leftarrow \text{Encap}(pk) ; \\
& m_1 \leftarrow \mathcal{U}(\mathcal{M}) ; \\
& b \leftarrow \mathcal{U}(\{0,1\}) ; \\
& b' \leftarrow \langle \mathscr{A}_2(s, c, m_b) \bowtie \text{Decap}(sk) \rangle ; \\
& \text{Ret}(b = b')
\end{aligned}
$$

**Figure 2: KEM-IND-CCA using our ITree language.**

$$
\begin{aligned}
[\![t]\!]^0 &\triangleq \mathbb{0} \\
[\![\text{Ret}(r)]\!]^{n+1} &\triangleq \mathbb{1}_r \\
[\![\text{Tau}(t)]\!]^{n+1} &\triangleq [\![t]\!]^n \\
[\![\text{Vis}(\text{Rnd}(X), k)]\!]^{n+1} &\triangleq \lambda r.\ \frac{1}{|X|} \sum_{x \in X} [\![k(x)]\!]^n(r) \\
[\![t]\!] &\triangleq \lim_{n \to \infty} [\![t]\!]^n
\end{aligned}
$$

**Figure 3: Probabilistic semantics for ITrees.**

decapsulation oracle ($+'$ denotes the disjoint union of event types). This adversary is modeled as a pair of probabilistic algorithms:

$$
\begin{aligned}
\mathscr{A}_1 &: \mathcal{PK} \to \text{itree } (\text{Dec} +' \text{Rnd}) \ \mathcal{S} \\
\mathscr{A}_2 &: \mathcal{S} \times C \times \mathcal{M} \to \text{itree } (\text{Dec} +' \text{Rnd}) \ \{0,1\},
\end{aligned}
$$

where $\mathcal{S}$ represents the adversary's state.

Figure 2 presents the standard game for KEM-IND-CCA. In the first stage, the challenger generates a key pair $pk$ and $sk$ and reveals the public key to the adversary $\mathscr{A}_1$. The adversary queries the decapsulation oracle (potentially many times, the interaction is denoted $\bowtie$), and writes to their state $s$. Note that the adversary can produce encapsulations at will, since the public key is known. Then, the challenger calls the encapsulation oracle to generate a message-ciphertext pair $m_0$ and $c$. The challenger then samples a random message $m_1$ and a bit $b$ uniformly, giving $m_b$ and $c$ to $\mathscr{A}_2$, which returns a bit $b'$. The adversary wins if the guess $b'$ is $b$ and $\mathscr{A}_2$ did not query the decapsulation oracle on the challenge ciphertext $c$: the adversary's *advantage* is defined as

$$
\text{Adv}_{\text{KEM-IND-CCA}}(\mathscr{C}, \mathscr{A}) \triangleq \left| \Pr\left[ \begin{matrix} \text{KEM-IND-CCA}_{\mathscr{C},\mathscr{A}} = 1 \text{ and} \\ c \notin \mathcal{L}(\mathscr{A}_2) \end{matrix} \right] - \frac{1}{2} \right|,
$$

where $\mathcal{L}(\mathscr{A}_2)$ is the set of ciphertexts that $\mathscr{A}_2$ queries Decap on.

## 2.3 Probabilistic Semantics for Games

Formalizing the adversary's advantage requires a semantics for our ITree language, defined below. This section defines a semantics for games that will allow us to define the adversary's advantage formally. Concretely, we first introduce a concrete representation of ITrees, define a generic probabilistic semantics for them, and finally use it on games.

*Interaction Trees.* Given a family of event types $\mathsf{E} : \text{Type} \to \text{Type}$ and a type of results R, the type itree E R represents a computation as a (potentially) infinite tree of all its behaviors, where leaf nodes are results R and non-leaf nodes are either silent steps or visible events E. Concretely, ITrees are defined coinductively as

$$
\text{itree E R} \ni t ::= \text{Ret}(r) \mid \text{Tau}(t) \mid \text{Vis}(\mathsf{A}, e, k),
$$

where $r : \mathsf{R}$, A is a type, $e : \mathsf{E\ A}$, and $k : \mathsf{A} \to \text{itree E R}$. The *return* constructor $\text{Ret}(r)$ labels the leaves of the tree: each leaf denotes a completed computation that yields the result $r$. Silent nodes $\text{Tau}(t)$ perform an unobservable transition before continuing as $t$ (i.e., there is only one branch). Visible event nodes $\text{Vis}(\mathsf{A}, e, k)$ trigger the event $e$, expect an answer $a : \mathsf{A}$, and continue as $k(a)$ (i.e., there is one branch per answer). Note that each Vis node has its own type A. We leave A implicit when it is clear from the context.

*Probabilistic Semantics.* Figure 3 presents the probabilistic interpretation of ITrees. Given a tree $t$ : itree Rnd R, we first define its $n$-step approximation semantics $[\![t]\!]^n : \mathsf{R} \to [0,1]$ (a distribution), and then take its limit. All zero-step approximations are the null distribution $\mathbb{0}$, which assigns zero probability to all results. All positive approximations of $\text{Ret}(r)$ are the unit distribution on the result $r$, denoted $\mathbb{1}_r$, which assigns probability one to $r$ and zero to all other values. The $(n+1)$-step approximation of $\text{Tau}(t)$ is the $n$-step approximation of $t$—note that this means that, as expected, diverging executions have probability zero. Then, the $(n+1)$-step approximation of $\text{Vis}(\text{Rnd}(X), k)$ accumulates the probabilities over all possible random samples $x \in X$ and divides by the number of such possible samples; technically, this clause implements a monadic bind where the first argument is the uniform distribution over the random space $X$. Finally, the probabilistic interpretation of $t$, denoted $[\![t]\!]$, is the limit over $n$ of $[\![t]\!]^n$ (which exists because $[\![t]\!]^n$ is increasing on $n$). This semantics is denotational—i.e., it is a distribution—as is the natural definition in pen-and-paper presentations. As expected, this semantics supports almost-sure termination (programs with termination probability one, e.g., rejection sampling, give rise to proper distributions) and assigns null probability to diverging computations (in such cases, the semantics is a *subdistribution* rather than a proper one; our formalization is generalized to this case as well).

*Semantics for Security Games.* The semantics of KEM-IND-CCA interprets assignments $x \leftarrow t\,; k$ with the ITree sequencing operator (i.e., the monadic bind), returns $\text{Ret}(r)$ with the return constructor, $\text{trigger}(e)$ as visible event nodes, and interactions $\langle t \bowtie o \rangle$ using the interp ITree operator (which replaces $t$'s events $\text{Dec}(c)$ with the result of calling the oracle $o$ on $c$). Crucially, the $\bowtie$ operator keeps track of all queries that $t$ performs to the oracle $o$. This allows us to define $\text{KEM-IND-CCA}_{\mathscr{C},\mathscr{A}}$ as an itree Rnd $(\{0,1\} \times C \times \mathcal{P}(C))$, where the first component is the correctness of the adversary's guess, the second one is the challenge ciphertext $c$, and the third one is the set of ciphertexts that $\mathscr{A}_2$ queried to the decapsulation oracle (i.e., corresponds to $\mathcal{L}(\mathscr{A}_2)$). Finally, the adversary's advantage is

$$
\text{Adv}_{\text{KEM-IND-CCA}}(\mathscr{C}, \mathscr{A}) \triangleq \left| \Pr\left[ \begin{matrix} [\![\text{KEM-IND-CCA}_{\mathscr{C},\mathscr{A}}]\!] = 1, c, \vec{q} \\ \text{and } c \notin \vec{q} \end{matrix} \right] - \frac{1}{2} \right|.
$$

## 2.4 Relating Security Games

Most security proofs are based on *reductionist arguments*, which reduce the security of one cryptographic system to the security of another system or a hardness assumption. In the case of compilers, the aim is to reduce the security of the compiled program to the source's, such that users need only reason about source programs

in their security proofs. Essentially, reductionist arguments state that attacks for the compiled program can be turned into attacks for the source program with acceptable advantage.

Our framework allows stating such reductions and proving them. These proofs fundamentally rely on the following result about the probabilistic semantics of equivalent ITrees.

Theorem 2.1 (). *Equivalent ITrees $t \approx t'$ give the same distribution, i.e.,* $\Pr[[\![t]\!] = \sigma] = \Pr[[\![t']\!] = \sigma]$ *for each* $\sigma$ *(that is,* $[\![t]\!] = [\![t']\!]$*).*

The equivalence relation is the standard one in the ITree library (weak bisimilarity, discussed in more detail in Section 5.2) and is what the Jasmin compiler achieves after our changes.

## 3 Interaction Trees Semantics of Jasmin

This section presents an interaction tree semantics for the Jasmin language and instantiates the notions of game-based security.

### 3.1 Background on the Jasmin Language

Jasmin [7, 9] is a framework for high-assurance high-speed cryptography. The main components of the framework are the Jasmin programming language and its compiler.

The language supports "assembly in the head," empowering programmers to write highly optimized code that remains amenable to formal analysis and verification. Concretely, the language features a combination of low-level constructs (such as vectorized instructions) and high-level constructs (such as structured control flow, e.g., conditionals, loops, and functions), and zero-cost abstractions (such as explicit variable names and functional arrays). The latter are called zero-cost because they introduce no run-time overhead; for example, functional array manipulations are always compiled to in-place memory operations that involve no copies. The core syntax of Jasmin from this perspective is as follows:

$$\begin{aligned}
\mathsf{Expr} \ni e &::= n \mid b \mid x \mid (e, \cdots, e) \mid \oplus(e) \mid x[e] \mid [e] \\
\mathsf{LVal} \ni \ell &::= x \mid x[e] \mid [e] \mid (\ell, \cdots, \ell) \\
\mathsf{Comm} \ni c &::= \mathbf{skip} \mid \ell \coloneqq e \mid \ell \coloneqq \mathbf{rand}(e) \mid \ell \coloneqq f(e) \\
&\quad \mid c\,;c \mid \mathbf{if}\ (e)\ \{c\}\ \{c\} \mid \mathbf{while}\ (e)\ \{c\}
\end{aligned}$$

Here, $e$ is an expression, $n$ is a natural number, $b$ is a boolean, $x \in \mathsf{Var}$ is a variable, $(e, \cdots, e)$ is a tuple of expressions, $\oplus$ is an operator (e.g., $+$, $\leq$, or $\wedge$), $\ell$ is a left-value, $x[e]$ is an array access to $x$ at position $e$, $[e]$ is a memory access dereferencing the address $e$, and $f$ is a function name.

### 3.2 Interaction-Tree and Probabilistic Semantics

This section presents a probabilistic semantics of Jasmin programs based on ITrees. The semantics is defined in several layers, as is standard in the ITree literature. First, we define the *call semantics*, which treats function calls, errors, and random sampling as uninterpreted events in the program (e.g., instead of executing the body of a callee, it triggers an event). Afterward, we interpret call events and obtain the *interprocedural semantics*, which executes the bodies of callees to answer call events. Lastly, we interpret the remaining events and achieve a *probabilistic semantics* that maps commands to distributions over states. This layered definition structure allows us to seamlessly move between levels of abstraction: we use specific layers to reason about relevant features of the semantics and treat other

$$\begin{aligned}
(\!|\mathbf{skip}|\!)^{\star}_{\sigma} &\triangleq \mathsf{Ret}(\sigma) \\
(\!|\ell \coloneqq e|\!)^{\star}_{\sigma} &\triangleq v \leftarrow (\!|e|\!)^{\mathsf{Expr}}_{\sigma}\,;\, (\!|\ell \leftarrow v|\!)^{\mathsf{LVal}}_{\sigma} \\
(\!|\ell \coloneqq \mathbf{rand}(e)|\!)^{\star}_{\sigma} &\triangleq n \leftarrow (\!|e|\!)^{\mathsf{Expr}}_{\sigma}\,;\, a \leftarrow \mathsf{trigger}(\mathsf{Rnd}(n))\,;\, (\!|\ell \leftarrow a|\!)^{\mathsf{LVal}}_{\sigma} \\
(\!|c\,;c'|\!)^{\star}_{\sigma} &\triangleq \sigma' \leftarrow (\!|c|\!)^{\star}_{\sigma}\,;\, (\!|c'|\!)^{\star}_{\sigma'} \\
(\!|\mathbf{if}\ (e)\ \{c_{\top}\}\ \{c_{\bot}\}|\!)^{\star}_{\sigma} &\triangleq b \leftarrow (\!|e|\!)^{\mathsf{Expr}}_{\sigma}\,;\, (\!|c_b|\!)^{\star}_{\sigma} \\
(\!|\mathbf{while}\ (e)\ \{c\}|\!)^{\star}_{\sigma} &\triangleq \mathsf{iter}\Big(\lambda\sigma.\, b \leftarrow (\!|e|\!)^{\mathsf{Expr}}_{\sigma}\,;\, W\Big)(\sigma) \\
&\qquad \text{where } W = \begin{cases} \sigma' \leftarrow (\!|c|\!)^{\star}_{\sigma}\,;\, \mathsf{Ret}(\mathsf{inl}(\sigma')) & \text{if } b, \\ \mathsf{Ret}(\mathsf{inr}(\sigma)) & \text{otherwise} \end{cases} \\
(\!|\ell \coloneqq f(e)|\!)^{\star}_{\sigma} &\triangleq v \leftarrow (\!|e|\!)^{\mathsf{Expr}}_{\sigma}\,; \\
&\qquad (m', v') \leftarrow \mathsf{trigger}(\mathsf{Call}(f, \sigma_m, v))\,; \\
&\qquad (\!|\ell \leftarrow v'|\!)^{\mathsf{LVal}}_{(m', \sigma_r)}
\end{aligned}$$

**(a) Call semantics. The** trigger **operator introduces a visible event node (e.g.,** $\mathsf{Rnd}(n)$**) and returns the answer (e.g.,** $a$**).**

$$\begin{aligned}
H^{c}(\mathsf{Call}(f, m, v)) &\triangleq \sigma \leftarrow (\!|f_{\mathsf{body}}|\!)^{\star}_{m, \{f_{\mathsf{arg}} \mapsto v\}}\,;\, \mathsf{Ret}(\sigma_m, \sigma_r(f_{\mathsf{res}})) \\
(\!|c|\!)_{\sigma} &: \mathsf{itree}\ (\mathsf{Err} +' \mathsf{Rnd})\ \mathsf{S} \triangleq \mathsf{interp_mrec}(H^{c}, (\!|c|\!)^{\star}_{\sigma}),
\end{aligned}$$

**(b) Interprocedural semantics. The** interp_mrec **operator answers every** Call **event with the** $H^{c}$ **handler (including those triggered by** $H^{c}$**).**

**Figure 4: Interaction Tree Semantics for Jasmin.**

features abstractly, enabling high reuse of code and proofs in generic lemmas of the compiler, in our program logic, and in our definitions of security. We now describe the layers relevant to this work.

The first layer is the call semantics, which employs three event families: Call for function calls, Err for errors, and Rnd for random sampling, denoted $(\!|c|\!)^{\star}_{\sigma} : \mathsf{itree}\ (\mathsf{Call} +' \mathsf{Err} +' \mathsf{Rnd})\ \mathsf{S}$ (where $c$ is a command, $\sigma \in \mathsf{S}$ a state). A state $\sigma \in \mathsf{S}$ is a pair of a variable map $r : \mathsf{Var} \to \mathsf{Val}$ (assigning values to variables) and a memory $m : \{0,1\}^{64} \to \{0,1\}^{8}$ (assigning bytes to addresses). The projections $\sigma_r$ and $\sigma_m$ are the variable map and memory of $\sigma$, respectively. The call semantics relies on a semantics of expressions $(\!|e|\!)^{\mathsf{Expr}}_{\sigma} : \mathsf{itree}\ \mathsf{Err}\ \mathsf{Val}$ (where $e$ is an expression) and left-values $(\!|\ell \leftarrow v|\!)^{\mathsf{LVal}}_{\sigma} : \mathsf{itree}\ \mathsf{Err}\ \mathsf{S}$ (where $v$ is a value), which are standard.

Figure 4a defines the call semantics. The **skip** command returns the state unchanged. An assignment evaluates the right-hand side and writes the result to the state—note that either step can trigger an Err event. A random sampling $\ell \coloneqq \mathbf{rand}(e)$ evaluates $e$ to a number $n$, triggers a $\mathsf{Rnd}(n)$ event (which expects $n$ uniformly sampled bytes) and writes them to $\ell$. (Note that this event is a special case of the one presented in Section 2.3, as Jasmin only supports sampling bytes.) The cases of sequencing and conditionals use the standard ITree monadic bind operator. While loops iteratively (potentially infinitely) execute the body with the iter operator. Finally, function calls evaluate their argument $e$, trigger a Call event, write the answer to $\ell$, and update the memory. Each Call event carries function name, memory, and argument value.

The second layer is the interprocedural semantics, which builds atop the call semantics by interpreting Call events. As is standard with ITrees, we define a *handler* for $\mathsf{Call}(f, m, v)$ events, which executes the body of the callee $f$ on an initial state with memory $m$ and arguments $v$, as shown in Figure 4b (there, $\{f_{\mathsf{arg}} \mapsto v\}$ is the

**Table 1: Semantics Used in This Paper.**

| Semantics | Symbol |
|---|---|
| Call | $(\!\lvert c \rvert\!)^{\star}_{\sigma}$ : itree (Call +′ Err +′ Rnd) S |
| Interprocedural | $(\!\lvert c \rvert\!)_{\sigma}$ : itree (Err +′ Rnd) S |
| Probabilistic | $[\![c]\!]_{\sigma}$ : S ∪ {$\sigma_{\text{err}}$} → [0, 1] |

variable map that contains $v$ as the argument of $f$). Figure 4b uses this handler and the mutual recursion operator interp_mrec to define the interprocedural semantics, consuming all Call events. The Err and Rnd events, on the other hand, remain uninterpreted, as effects of the program.

Lastly, we handle Err events by returning a distinguished error state $\sigma_{\text{err}}$, obtaining an itree Rnd (S ∪ {$\sigma_{\text{err}}$}). Then, we use the semantics from Figure 3 to obtain a semantics that maps every command $c$ and state $\sigma$ to a distribution over states or errors $[\![c]\!]_{\sigma}$ : S ∪ {$\sigma_{\text{err}}$} → [0, 1]. Table 1 summarizes the semantics notions used in this paper, each defined in terms of the one above.

*Safety.* A Jasmin program is *safe* (relative to some precondition) if for every function $f$ and state $\sigma$ satisfying the precondition for $f$, its semantics triggers no Err events and its results are fully initialized. Specifically, the latter condition requires that every cell of returned arrays is set to a value—Jasmin allows programs to take and return arrays with some cells uninitialized. Consequently, for safe Jasmin programs, the probabilistic semantics is simply a distribution over states $[\![c]\!]_{\sigma}$ : S → [0, 1]. In contrast to the original Jasmin semantics, termination is not a requirement for safety; in particular, almost-surely-terminating programs like ML-KEM can be safe and the guarantees of compiler correctness apply to them.

## 3.3 Game-Based Security of Jasmin Programs

Using the semantics presented above, we can instantiate the framework from Section 2 and give formal definitions of KEM-IND-CCA security for Jasmin KEMs. The challenger induced by the Jasmin KEM program $P$ is denoted $\mathscr{C}_P$ and defined as

$$\begin{aligned}
\text{GenKey}() &\triangleq \sigma' \leftarrow (\!\lvert P(\texttt{genkey}) \rvert\!)_{m,\emptyset}\,;\text{Ret}(\sigma'(\texttt{pk}), \sigma'(\texttt{sk}))\\
\text{Encap}(pk) &\triangleq \sigma' \leftarrow (\!\lvert P(\texttt{encap}) \rvert\!)_{m,\{\texttt{pk}\mapsto pk\}}\,;\text{Ret}(\sigma'(\texttt{msg}), \sigma'(\texttt{ct}))\\
\text{Decap}(sk, c) &\triangleq \sigma' \leftarrow (\!\lvert P(\texttt{decap}) \rvert\!)_{m,\{\texttt{sk}\mapsto sk,\texttt{ct}\mapsto c\}}\,;\text{Ret}(\sigma'(\texttt{msg}))
\end{aligned}$$

We write genkey, encap, and decap for the names of the respective functions in the program, pk, sk, ct, and msg for those of the variables, and $m$ for an initial memory on which the program is safe.

The key generation algorithm first executes the body of the genkey function in $P$ on the initial memory $m$ and the empty variable map $\emptyset$, and then returns the values of pk and sk in the final state. The encapsulation algorithm executes encap on a state that maps the variable pk to the given value $pk$ and returns the final values of msg and ct. Similarly, the decapsulation algorithm executes decap on a state that maps the variables sk and ct to the values $sk$ and $c$, returning the final value of msg. The ITrees for these functions have only Rnd events, coinciding with the definitions in Section 2.1—we prove that no error events Err can happen because we consider safe programs. Thus, queries are handled by ⋈ in a similar way as function calls (i.e., running the code of Decap). Finally, we define the advantage of an adversary $\mathscr{A}$ against a Jasmin KEM program $P$ as $\text{Adv}_{\text{KEM-IND-CCA}}(\mathscr{C}_P, \mathscr{A})$ (using the definition from Section 2.3).

## 3.4 Game-Based Security of Assembly Programs

The semantics of assembly programs has a similar signature to the one for Jasmin programs, but it operates on *assembly states*, which comprise: a register map, a memory, a program counter, and the name of the function currently executing. As expected, the semantics is defined as the repeated application of a small-step transition function from assembly states to assembly states.

The security notion for assembly programs has two key differences with the one for Jasmin. First, array arguments are not given as values but rather as pointers to memory; secondly, the initial memory is initialized with the contents of such arguments. Reading the results from final states changes in a similar way. Concretely, in the case of a KEM, the challenger for a compilation $Q$, denoted $\mathscr{C}_Q$, is parameterized by four pointers ppk, psk, pct, and pmsg (which are 64-bit words in x86-64) corresponding to the variables used in $\mathscr{C}_P$. Thus, in Encap, instead of initializing a variable map with the array $pk$, we write $pk$ to memory at position ppk and initialize the register corresponding to the variable pk to ppk.

# 4 Correctness and KEM-IND-CCA Preservation

This section overviews the Jasmin compiler, states the new compiler correctness theorem and compares it with the existing one, and presents its KEM-IND-CCA security preservation result. Afterward, we discuss the application of our results to Jasmin's ML-KEM implementation. Finally, we generalize our results to all game-based security properties, leading to the main result of this work: that the Jasmin compiler preserves game-based security (Theorem 4.5).

The Jasmin compiler is a moderately optimizing compiler that is designed to be predictable and provide a large degree of control to the programmer; thus, its primary purpose is to eliminate the high-level constructs and zero-cost abstractions described in Section 3.1. The compiler performs a reduced number of optimizations, such as dead code elimination and memory reuse (to minimize stack usage). Figure 5 presents the passes of the Jasmin compiler. Ellipses correspond to the different representations of programs, rectangles to transformations, and dotted rectangles to checkers—i.e., passes that leave the program unchanged but may reject it. Red boxes correspond to trusted steps, green ones to those verified in Rocq directly, and blue ones to steps verified using translation validation. The passes inside the top dashed gray box (second to sixth rows) are the compiler *front-end*, whereas the ones inside the box below (seventh row) are the *back-end*. Section B presents a summary of the functionality of each pass. The compiler supports the x86-64, ARMv7, and RISC-V architectures.

## 4.1 Compiler Correctness Statement

We updated the correctness statement and proofs of the compiler to use the new ITree semantics. Crucially, *we did not modify the functionality of the compiler*, i.e., the compiler before and after our changes generates the same assembly; this means that we proved a stronger statement while retaining the functionality. The new

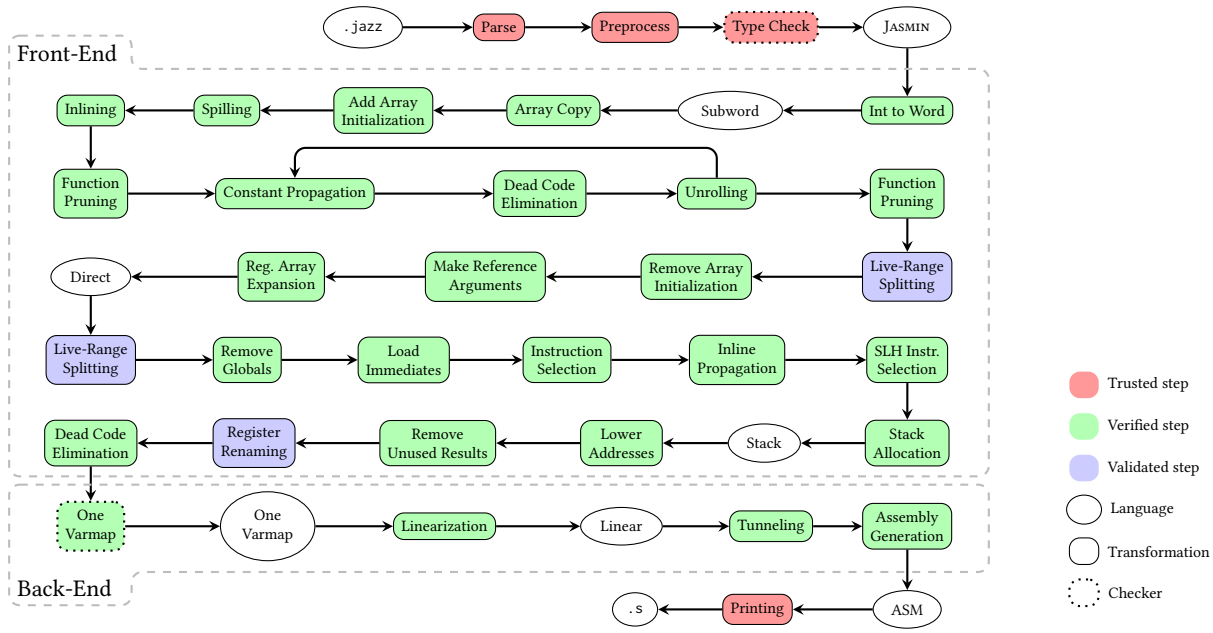


**Figure 5: Passes and intermediate languages of the Jasmin compiler.**

correctness theorem for the compiler is stated in terms of a bisimulation (including Rnd events) between the semantics of the source program and its compilation, as follows.

Theorem 4.1 (Compiler Correctness 🏆). *Given a program $P$, its compilation $Q$, any function $f$, and source and target states $\sigma$ and $\tau$,*

$$\sigma \preccurlyeq \tau \implies \langle\!|P(f)|\!\rangle_\sigma \approx_{\preccurlyeq} \langle\!|Q(f)|\!\rangle_\tau$$

*under the following conditions:*

1. *The source program $P$ is safe.*
2. *The initial target memory has enough allocated fresh memory for the stack of each function (the compiler statically determines how much stack space each function needs).*
3. *Each writable input pointer is disjoint from all other writable or read-only input pointers and from the* `.data` *section.*

Bisimulation, denoted $\approx$, means that the two ITrees are the same except for a finite number of silent steps. It is indexed by a relation on results, $\preccurlyeq$ in this case, that relates the results (i.e., the leaves). It is the standard bisimulation notion for LTSs and verified compilers, and it is provided by the ITree library (we define it in Section A).

Intuitively, the above theorem states that running $f$ on related source-target states $\sigma \preccurlyeq \tau$ produces the same behaviors (Rnd events) and, furthermore, that the final states (if they exist) are also related. We define bisimulation precisely in Section 5.2, and compare it with the usual forward and backward compiler simulations in Section 4.2.

The refinement relation $\preccurlyeq$ involves several well-formedness conditions (e.g., that array values in the source correspond to pointers in the target) and relates values with the usual *more defined than* relation. Specifically, a target value is more defined than a source value if either they are equal or the latter is undefined. In the case of arrays, the simulation requires that a pointer in the target points to a region that is more defined than the corresponding array in the source.

## 4.2 Comparison with Original Statement

The original semantics of Jasmin programs is a big-step operational semantics of the form $\langle c, \sigma\rangle \Downarrow \sigma'$, stating that running a command $c$ on a state $\sigma$ terminates without errors and reaches a state $\sigma'$ [7, 9]. Random sampling commands consumed an infinite random tape. Compiler correctness was stated as a forward simulation between a source command $P$ and its compilation $Q$: for every function $f$, source states $\sigma$ and $\sigma'$, and target state $\tau$,

$$\langle P(f), \sigma\rangle \Downarrow \sigma' \wedge \sigma \preccurlyeq \tau \implies \exists \tau'.\, \langle Q(f), \tau\rangle \Downarrow \tau' \wedge \sigma' \preccurlyeq \tau',$$

where $\preccurlyeq$ is the relation presented in Section 4.1 with an additional clause ensuring equal random tapes.

Consequently, the original compiler correctness statement ensures that safe and terminating Jasmin programs are compiled to safe and terminating programs with equal results. However, it gives no immediate guarantees for executions that diverge or for the probabilistic behavior (i.e., as distributions) of programs.

Our work improves over the original compiler correctness in two key directions. First, our semantics covers diverging and almost-surely terminating executions, and is expressed directly as a distribution rather than requiring a tape. Secondly, while our proofs follow the natural forward reasoning pattern, they directly establish a bisimulation rather than a forward simulation, with no need for flipping diagrams.

An alternative strategy to overcome the challenges above would have been to use a small-step operational semantics, with a randomness tape or oracle, producing an infinite trace. Interaction trees dispense us from going through such a semantics, which comes with its difficulties. First, since our end goal of preservation of cryptographic security requires a distributional semantics for both source and target languages, using a trace semantics would require gluing the compiler correctness statement with distributional semantics

for both Jasmin and assembly. Additionally, nontermination would require infinite tapes, making the semantics continuous and requiring further glue to recover the desired cryptographic security statement. In contrast, ITrees provide a unified framework to define the semantics, verify the compiler, and reason about probabilistic behavior and adversarial interactions.

## 4.3 Main Theorem

This section leverages the compiler correctness result and the framework from Section 2 to show that the Jasmin compiler preserves KEM-IND-CCA security. We use Theorem 2.1 to prove a KEM-IND-CCA security reduction for the Jasmin compiler. That is, we show that Jasmin preserves KEM-IND-CCA security when compiling a KEM, as stated in the following theorem.

Theorem 4.2 (KEM-IND-CCA Security Preservation). *Given a source program $P$ with compilation $Q$ and an adversary $\mathscr{A}$, we have*

$$\mathrm{Adv}_{\text{KEM-IND-CCA}}(\mathscr{C}_P, \mathscr{A}) = \mathrm{Adv}_{\text{KEM-IND-CCA}}(\mathscr{C}_Q, \mathscr{A}),$$

*under the compiler correctness conditions from before.*

This theorem is an exact reduction: advantages are equal, and the adversary is the same.

## 4.4 Application to ML-KEM

Almeida et al. [6] use EasyCrypt to prove that a highly optimized Jasmin implementation of ML-KEM (denoted $\mathsf{ML\text{-}KEM}_{\text{Jasmin}}$) is provably secure under the Hashed Module variant of the Learning with Errors assumption (MLWE) [69]. By combining the security proof with an adaptation of the functional correctness proof in Almeida et al. [10] (the original uses CRYSTALS-Kyber [35]), they establish the following theorem.

Theorem 4.3 (Provable Security of $\mathsf{ML\text{-}KEM}_{\text{Jasmin}}$). *For every adversary $\mathscr{A}$ that makes at most $d$ queries to the hash function and executes in time at most $T$,*

$$\mathrm{Adv}_{\text{KEM-IND-CCA}}(\mathsf{ML\text{-}KEM}_{\text{Jasmin}}, \mathscr{A}) \leq \epsilon(T, d).$$

Here $\epsilon$ is a complex expression involving (1) the correctness bound of $\mathsf{ML\text{-}KEM}_{\text{Jasmin}}$, i.e., the probability that decapsulation produces an incorrect output; (2) the advantage of an adversary executing in time at most $T'$ (where $T'$ is close to $T$) against Hashed MLWE; and (3) pseudorandomness advantages of variants of SHAKE and of the key generation smoothing function used in the implementation. The full definition is in [6, Theorem 6]. Our work yields a security proof for *the compilation* of ML-KEM (denoted $\mathsf{ML\text{-}KEM}_{\text{ASM}}$), as stated in the following corollary.

Corollary 4.4 (Provable Security of $\mathsf{ML\text{-}KEM}_{\text{ASM}}$). *For every adversary $\mathscr{A}$ that makes at most $d$ queries to the hash function and executes in time at most $T$,*

$$\mathrm{Adv}_{\text{KEM-IND-CCA}}(\mathsf{ML\text{-}KEM}_{\text{ASM}}, \mathscr{A}) \leq \epsilon(T, d).$$

Naturally, since Theorem 4.3 is mechanized in EasyCrypt, the TCB of Theorem 4.4 also includes the assumption that the former proof implies that the Jasmin program is KEM-IND-CCA secure.

## 4.5 Generalization to Game-Based Security

As mentioned in the introduction, our Rocq development deals with a general notion of game-based security; thus, all preceding theorems are proven by instantiating this general framework. The framework is based on oracle systems [23], which model interactive games between adaptive adversaries and a cryptographic scheme through oracle queries. They provide a unified representation of all common cryptographic properties—e.g., KEM-IND-CCA and SUF-CMA—and can capture different models of cryptography such as the standard, the random oracle, and the ideal ciphertext models.

An oracle system is a family of stateful probabilistic algorithms indexed by natural numbers. A pair of functions $\mathsf{I} : \mathbb{N} \to \mathsf{Type}$ and $\mathsf{O} : \mathbb{N} \to \mathsf{Type}$ give the input and output types of each oracle, respectively. Oracle queries are events $\mathsf{Que}(n, i)$ such that $n : \mathbb{N}$ is an oracle, $i : \mathsf{I}_n$ is an input, and the expected answer is of type $\mathsf{O}_n$. The challenger in this setting gives an implementation for each oracle, taking an input of type $\mathsf{I}_n$ and returning an output of type $\mathsf{O}_n$, and updating an oracle memory $\mathbb{M}$. On the other hand, the adversary is a probabilistic algorithm that performs oracle queries and returns in the unit type. In symbols,

$$\mathscr{C} : (n : \mathbb{N}) \to (\mathsf{I}_n \times \mathbb{M}) \to \mathsf{itree}\ \mathsf{Rnd}\ (\mathsf{O}_n \times \mathbb{M}),$$
$$\mathscr{A} : \mathsf{itree}\ (\mathsf{Que} +' \mathsf{Rnd})\ \mathsf{Unit}.$$

A game executes the adversary and interprets Que events with the appropriate oracles of the challenger. The adversary wins the game if the trace of query-answer pairs satisfies a predicate called the *winning condition* $\mathsf{win} : (n, \mathsf{I}_n, \mathsf{O}_n)^* \to \{0, 1\}$.

Oracle systems have been used to model a variety of cryptographic schemes and properties [23, 33, 78]. As an illustrative example, the KEM-IND-CCA oracle system for a given KEM consists of four oracles: (1) an initialization oracle $o_I$ (with $\mathsf{I}_{o_I} = \mathsf{Unit}$ and $\mathsf{O}_{o_I} = \mathcal{PK}$), which uses GenKey to generate a key pair, stores it in the memory, and returns the public key; (2) a challenge oracle $o_C$ (with $\mathsf{I}_{o_C} = \mathsf{Unit}$ and $\mathsf{O}_{o_C} = \mathcal{M} \times \mathcal{C}$), which uses Encap with the stored public key, samples a random bit and message, stores the bit, and returns the chosen message and ciphertext; (3) a decapsulation query oracle $o_D$ (with $\mathsf{I}_{o_D} = \mathcal{C}$ and $\mathsf{O}_{o_D} = \mathcal{M}$), which uses Decap with the stored secret key to decapsulate the ciphertext and returns its result; and (4) a finalize oracle $o_F$ (with $\mathsf{I}_{o_F} = \{0, 1\}$ and $\mathsf{O}_{o_F} = \mathsf{Unit}$), which receives the adversary's guess. The winning condition checks that the adversary's guess is correct, that the interactions follow the expected order, and that the adversary did not query the decapsulation oracle on the challenge ciphertext.

Using our framework for oracle systems, we can generalize Theorem 4.2 to all game-based security properties, as follows.

Theorem 4.5 (Security Preservation). *Given a source program $P$ with compilation $Q$ and an adversary $\mathscr{A}$, we have that*

$$\Pr[\mathscr{A}\ \textit{wins against}\ \mathscr{C}_P] = \Pr[\mathscr{A}\ \textit{wins against}\ \mathscr{C}_Q],$$

*where $\mathscr{C}_P$ and $\mathscr{C}_Q$ are the oracle systems induced by $P$ and $Q$, respectively, and we say that an adversary "wins against" a challenger if their interaction produces a trace that satisfies the winning condition.*

The above theorem is proven directly from source to target, relying only on the statement of compiler correctness from Theorem 4.1 and generic ITree properties (in particular, it is independent of the compiler and its passes).

## 5 Relational Hoare Logic for Interaction Trees

In this section, we present the key proof technique used to verify the compiler in our new setting: a relational Hoare logic for ITree-based program semantics. We first present standard rules and then discuss specialized versions for compiler correctness proofs.

### 5.1 Tuples and Validity

Relational Hoare logic tuples relate two commands with respect to pre- and postconditions as follows.

*Definition 5.1 (Valid Command Tuple).* A command tuple is valid, written $I_P, I_Q \models c_1 \sim c_2 : \Phi \Rightarrow \Psi$, when for every pair of states $\sigma_1$ and $\sigma_2$,

$$\sigma_1 \ \Phi \ \sigma_2 \implies \overset{u}{\approx}_{I_P, I_Q, \Psi} (\!| c_1 |\!)^{\star}_{\sigma_1} (\!| c_2 |\!)^{\star}_{\sigma_2},$$

where $c_1$ and $c_2$ are commands, $\Phi, \Psi : \mathsf{S} \to \mathsf{S} \to \mathsf{Prop}$ are state pre- and postconditions, and $I_P$ and $I_Q$ form a *handler contract*, where $I_P : \forall \mathsf{A}_1 \mathsf{A}_2.\, \mathsf{E}_1\ \mathsf{A}_1 \to \mathsf{E}_2\ \mathsf{A}_2 \to \mathsf{Prop}$ is an *event precondition* and $I_Q : \forall \mathsf{A}_1 \mathsf{A}_2.\, \mathsf{E}_1\ \mathsf{A}_1 \to \mathsf{A}_1 \to \mathsf{E}_2\ \mathsf{A}_2 \to \mathsf{A}_2 \to \mathsf{Prop}$ an *event postcondition*.

Intuitively, a command tuple is valid when running the commands $c_1$ and $c_2$ on $\Phi$-related states gives equivalent behaviors until either $c_1$ performs undefined behavior or both commands terminate with $\Psi$-related states. This equivalence of behaviors, denoted $\overset{u}{\approx}$ and defined in the next section, lifts the standard notion of bisimulation to ITrees with events (Call, Err, and Rnd in this case) and undefined behavior. It uses $I_P$ to relate events performed by the two commands and $I_Q$ to relate the environment's answers to said events. The judgment extends to programs as follows.

*Definition 5.2 (Valid Program Tuple).* A program tuple is valid, written $I_P, I_Q \models P_1 \sim P_2 : \Phi \Rightarrow \Psi$ when for every pair of functions $f$ and $g$ and states $\sigma_1$ and $\sigma_2$,

$$\sigma_1 \ \Phi_{f,g} \ \sigma_2 \implies \overset{u}{\approx}_{I_P, I_Q, \Psi_{f,g}} (\!| f_{\text{body}} |\!)_{\sigma_1} (\!| g_{\text{body}} |\!)_{\sigma_2},$$

where $P_1$ and $P_2$ are programs, $\Phi_{f,g}, \Psi_{f,g} : \mathsf{S} \to \mathsf{S} \to \mathsf{Prop}$ are the pre- and postconditions for functions (indexed by function names $f$ and $g$), $I_P$ and $I_Q$ form the handler contract.

Intuitively, a program tuple establishes that running the bodies of a pair of functions $f \in P_1$ and $g \in P_2$ on $\Phi_{f,g}$-related states produces equivalent behaviors forever or until either $f$ performs undefined behavior or both functions terminate with $\Psi_{f,g}$-related states.

In the rest of the paper, we use $\models e_1 \sim e_2 : \Phi \Rightarrow \psi$ (where $e_1$ and $e_2$ are expressions, $\Phi$ relates states, and $\psi$ relates values) to express that evaluating $e_1$ and $e_2$ in $\Phi$-related states yields $\psi$-related values. We use $\models \ell_1 \sim \ell_2 : \Phi/\phi \Rightarrow \Psi$ (where $\Phi$ and $\Psi$ relate states and $\phi$ values) when writing $\phi$-related values to $\Phi$-related states yields $\Psi$-related states.

### 5.2 Equivalence up-to-Cutoff

This section defines the relation $\overset{u}{\approx}$ needed by RHL tuples. We first introduce the standard notion of bisimulation from the ITree library and then our generalization.

Weak bisimulation, denoted $\approx_\phi$ and sometimes called equivalence up-to-tau, is a widespread notion of equivalence between ITrees that relates computations that differ only by a finite number of silent steps on either side and yield $\phi$-related results [80].

$$\textsc{Ret}\ \frac{r_1\ \phi\ r_2}{\mathsf{Ret}(r_1) \overset{e/u}{\approx} \mathsf{Ret}(r_2)} \qquad \textsc{Tau}\ \frac{t_1 \overset{e/u}{\approx} t_2}{\mathsf{Tau}(t_1) \overset{e/u}{\approx} \mathsf{Tau}(t_2)} \qquad \textsc{Cut}_L\ \frac{C_1(e_1)}{\overset{u}{\approx} \mathsf{Vis}(e_1, k_1)\, t_2}$$

$$\textsc{Cut}_R\ \frac{C_2(e_2)}{\overset{u}{\approx} t_1 \mathsf{Vis}(e_2, k_2)} \qquad \textsc{Tau}_L\ \frac{t_1 \overset{e/u}{\approx} t_2}{\mathsf{Tau}(t_1) \overset{e/u}{\approx} t_2} \qquad \textsc{Tau}_R\ \frac{t_1 \overset{e/u}{\approx} t_2}{t_1 \overset{e/u}{\approx} \mathsf{Tau}(t_2)}$$

$$\textsc{Vis}\ \frac{e_1\ I_P\ e_2 \qquad \forall a_1\, a_2.\, (e_1, a_1)\ I_Q\ (e_2, a_2) \implies k_1(a_1) \overset{e/u}{\approx} k_2(a_2)}{\mathsf{Vis}(e_1, k_1) \overset{e/u}{\approx} \mathsf{Vis}(e_2, k_2)}$$

**Figure 6: Bisimulation relations $\overset{e}{\approx}$ and $\overset{u}{\approx}$. The boxed rules are only for $\overset{u}{\approx}$, all others (with $\overset{e/u}{\approx}$) are valid for both $\overset{e}{\approx}$ and $\overset{u}{\approx}$.**

Weak bisimulation is too strong for many settings, as it requires events in the ITrees to match exactly. Consequently, the ITree library also provides *heterogeneous equivalence*, denoted $\overset{e}{\approx}_{\phi, I_P, I_Q}$, a generalization with two relations for events [32, 73]. It relates ITrees that depend on distinct return types (denoted $\mathsf{R}_1$ and $\mathsf{R}_2$, related by $\phi$) and distinct event families (denoted $\mathsf{E}_1$ and $\mathsf{E}_2$, related with $I_P$ and $I_Q$). It is a coinductive-inductive relation in the style of weak bisimulation.

Figure 6 defines $\overset{e}{\approx}$ (we distinguish inductive and coinductive uses of the premise with single and double bars, respectively [55]). The relation is parameterized by a relation $\phi : \mathsf{R}_1 \to \mathsf{R}_2 \to \mathsf{Prop}$, an event precondition $I_P : \forall \mathsf{A}_1 \mathsf{A}_2.\, \mathsf{E}_1\ \mathsf{A}_1 \to \mathsf{E}_2\ \mathsf{A}_2 \to \mathsf{Prop}$, and its postcondition $I_Q : \forall \mathsf{A}_1 \mathsf{A}_2.\, \mathsf{E}_1\ \mathsf{A}_1 \to \mathsf{A}_1 \to \mathsf{E}_2\ \mathsf{A}_2 \to \mathsf{A}_2 \to \mathsf{Prop}$.

The Ret rule relates any pair of $\phi$-related return values and the Tau rule relates two ITrees that perform a synchronized silent step. The $\textsc{Tau}_L$ and $\textsc{Tau}_R$ rules rely on their hypothesis *inductively* rather than coinductively, ensuring that related ITrees can differ only up to a *finite* number of silent transition steps. The Vis rule requires that, given answer types $\mathsf{A}_1$ and $\mathsf{A}_2$ and events $e_1 : \mathsf{E}_1\ \mathsf{A}_1$ and $e_2 : \mathsf{E}_2\ \mathsf{A}_2$, the events are related by $I_P$. The continuations $k_1(a_1)$ and $k_2(a_2)$ must also be in relation, assuming that the answers $a_1$ and $a_2$ satisfy the event postcondition $I_Q$.

*Our Generalization.* Heterogeneous equivalence is still too strong for compiler correctness proofs, which crucially rely on reasoning about undefined behavior on one side of the relation. Compiler correctness typically relates the semantics of a source program to its compilation's *as long as the source program is safe*. Assuming safety is often essential in compiler proofs, as it allows us to disregard invalid behaviors that need not be preserved. Additionally, safety also encodes important invariants of the compiler passes (e.g., if the compiler introduces a memory access, that access must be in-bounds) that subsequent passes rely upon.

As a result, our work further generalizes heterogeneous equivalence to support error handling in terms of undefined behavior. It relies on the notion of *cutoff event*, on either side of the relation, that allows the relation to hold trivially from that point, regardless of the continuation. Figure 6 defines the new relation $\overset{u}{\approx}$, called equivalence up-to-cutoff, as an extension of $\overset{e}{\approx}$ with two additional boolean predicate parameters, $C_i : \forall \mathsf{A}.\, \mathsf{E}_i\ \mathsf{A} \to \mathtt{bool}$ for $i = 1, 2$, which decide

$$\textsc{Assign}\quad \frac{\models e_1 \sim e_2 : \Phi \Rightarrow \phi \qquad \models \ell_1 \sim \ell_2 : \Phi/\phi \Rightarrow \Psi}{\vdash \ell_1 \coloneqq e_1 \sim \ell_2 \coloneqq e_2 : \Phi \Rightarrow \Psi}$$

$$\textsc{While}\quad \frac{\models e_1 \sim e_2 : \Phi \Rightarrow = \qquad \vdash c_1 \sim c_2 : \Phi \wedge e_1 \Rightarrow \Phi}{\vdash \textbf{while}\ (e_1)\ \{c_1\} \sim \textbf{while}\ (e_2)\ \{c_2\} : \Phi \Rightarrow \Phi \wedge \neg e_1}$$

$$\textsc{Rand}\quad \frac{\models e_1 \sim e_2 : \Phi \Rightarrow I_P \qquad \models \ell_1 \sim \ell_2 : \Phi/I_Q \Rightarrow \Psi}{\vdash \ell_1 \coloneqq \textbf{rand}(e_1) \sim \ell_2 \coloneqq \textbf{rand}(e_2) : \Phi \Rightarrow \Psi}$$

$$\textsc{Call}\quad \frac{\models e_1 \sim e_2 : \left(\Phi_r \wedge I_P^m\right) \Rightarrow I_P^v \qquad \models \ell_1 \sim \ell_2 : \left(\Phi_r \wedge I_Q^m\right)/I_Q^v \Rightarrow \Psi}{\vdash \ell_1 \coloneqq f(e_1) \sim \ell_2 \coloneqq g(e_2) : \left(\Phi_r \wedge I_P^m\right) \Rightarrow \Psi}$$

$$\textsc{Assign}_L\quad \frac{\forall \sigma_1 \sigma_1' \sigma_2.\ \sigma_1\ \Phi\ \sigma_2 \wedge (\!|\ell \coloneqq e|\!)^\star_{\sigma_1} \approx \mathrm{Ret}(\sigma_1') \implies \sigma_1'\ \Psi\ \sigma_2}{\vdash \ell \coloneqq e \sim \textbf{skip} : \Phi \Rightarrow \Psi}$$

$$\textsc{Trans}\quad \frac{I_{P1}, I_{Q1} \vdash c_1 \sim c_2 : \Phi_1 \Rightarrow \Psi_1 \qquad I_{P2}, I_{Q2} \vdash c_2 \sim c_3 : \Phi_2 \Rightarrow \Psi_2}{I_{P1} \circ I_{P2}, I_{Q1} \circ I_{Q2} \vdash c_1 \sim c_3 : \Phi_1 \circ \Phi_2 \Rightarrow \Psi_1 \circ \Psi_2}$$

$$\textsc{Rec}\quad \frac{I_P \cup \Phi, I_Q \cup \Psi \vdash f_{\text{body}} \sim g_{\text{body}} : \Phi_{f,g} \Rightarrow \Psi_{f,g} \quad \text{for every } f \in P_1, g \in P_2}{I_P, I_Q \vdash P_1 \sim P_2 : \Phi \Rightarrow \Psi}$$

**Figure 7: Relational Hoare Logic.**

which events are cutoffs on each side of the relation. The $\textsc{Cut}_L$ rule relates $\mathsf{Vis}(e,k) : \mathsf{itree}\ \mathsf{E}_1\ \mathsf{R}_1$ with any $t : \mathsf{itree}\ \mathsf{E}_2\ \mathsf{R}_2$ whenever $e$ is a cutoff event on the left-hand side; $\textsc{Cut}_R$ is symmetric. When there are no cutoffs on either side, $\overset{u}{\approx}$ is equivalent to $\overset{e}{\approx}$.

## 5.3 Proof Rules

Relational Hoare logic rules typically come in three flavors: *two-sided* rules, *one-sided* rules, and *structural* rules. This section briefly discusses each group in turn (we omit the Skip, Seq, and Case rules here, which are standard and included in Section C). As usual, $I_P, I_Q \vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi$ (resp. $I_P, I_Q \vdash P_1 \sim P_2 : \Phi \Rightarrow \Psi$) means that an RHL command (resp. program) tuple is derivable. We leave $I_P$ and $I_Q$ implicit when they are clear from the context.

*Two-Sided Rules.* The first four rules of Figure 7 are two-sided rules. The Assign rule states that two assignments are related if evaluating their expressions $e_1$ and $e_2$ under the assumption $\Phi$ yields $\phi$-related values, and writing such values in $\ell_1$ and $\ell_2$ ensures $\Psi$. The While rule is *synchronous*, i.e., it requires that the guards $e_1$ and $e_2$ evaluate to the same value, hence the postcondition for the expressions is equality. The formula $\Phi$ acts as a loop invariant, as in Hoare logic.

The Rand rule deals with random sampling. It requires that the arguments $e_1$ and $e_2$ satisfy the handler precondition $I_P$ (for Rnd events), and that the handler postcondition $I_Q$ ensures the postcondition $\Psi$ on the states after writing the answers to $\ell_1$ and $\ell_2$.

Finally, the Call rule relates two function calls. The specification for functions is given by the handler contract $I_P$ and $I_Q$ for Call events: it establishes the (relational) pre- and postconditions for calling each pair of functions. As such, $I_P$ and $I_Q$ are relations between pairs of lists of values and memories. Without loss of generality, we split them into $I_P^v$ and $I_P^m$, and $I_Q^v$ and $I_Q^m$, for the relations values and memories, respectively.

The first premise of the Call rule states that, given a precondition on variable maps $\Phi_r$, the arguments $e_1$ and $e_2$ must satisfy the functions' precondition $I_P^v$. The second premise asserts the postcondition $\Psi$ after writing the results in states where *the initial* precondition on variable maps $\Phi_r$ holds, as well as the functions' postcondition $I_Q$. The precondition $\Phi_r$ holds after writing results because the callee executes on its own local variable map, leaving the caller's unchanged.

*One-Sided Rules.* These rules relate structurally dissimilar programs—often, an event-free command with **skip**. The $\textsc{Assign}_L$ rule is an example: it relates an assignment on the left-hand side with **skip** on the right-hand side. The premise states that the postcondition $\Psi$ must hold after an assignment on the state on the left-hand side.

*Structural Rules.* Our logic satisfies many standard structural rules of program logics, such as reflexivity, consequence, and transitivity. For example, the Trans rule allows us to compose judgments transitively. We use $\circ$ for the composition of two relations, defined as $x\ (R \circ R')\ z \triangleq \exists y.\ x\ R\ y \wedge y\ R'\ z$.

*Recursion and Programs.* Figure 7 presents the Rec rule to relate two programs. The rule lifts and generalizes the classical recursion rule in Hoare logic [11] to the relational setting. As before, we write $f_{\text{body}}$ and $g_{\text{body}}$ for the bodies of $f$ and $g$ in $P_1$ and $P_2$, respectively. In the premise, we extend the contract to $I_P \cup \Phi$ and $I_Q \cup \Psi$, which relates Call events using $\Phi$ and $\Psi$, and other events using $I_P$ and $I_Q$. The rule requires proving that *the bodies* of every pair of functions $f$ and $g$ satisfy their relational specification *assuming that all function calls satisfy theirs*. Note that this allows reasoning about mutually recursive functions.

As a result, two programs are related if all pairs of bodies satisfy their relational specification. While the premise of this rule involves only the call semantics, the soundness theorem below shows that its conclusion relates the interprocedural semantics of functions, allowing us to reason abstractly about calls.

*Soundness.* The following theorem establishes the soundness of our program logic, relating the call and interprocedural semantics.

Theorem 5.3 (Soundness). *Provable judgments are valid, i.e.,* $I_P, I_Q \vdash P_1 \sim P_2 : \Phi \Rightarrow \Psi$ *implies* $I_P, I_Q \models P_1 \sim P_2 : \Phi \Rightarrow \Psi$.

# 6 Key Proof Techniques for Selected Passes

This section discusses the key techniques used to verify the compiler. Recall that the Jasmin compiler operates over seven different languages. The front-end has four internal languages: Jasmin (the source), Subword, Direct, and Stack (see Figure 5). These languages have the same syntax but different semantics, i.e., a language construct such as a function call has different effects, becoming lower-level down the compilation chain. Nevertheless, their semantics are instances of a general one, parameterized over the effects of certain constructs, such as function calls. The back-end begins with the Stack language and comprises three other languages: One Varmap,

Linear and ASM. All languages are defined in layers using ITrees; in particular, all have a layer with error and random events.

We verify the front-end using RHL: this section illustrates how RHL is used in the proofs of three such passes and distill three key relevant features of the logic—which are not specific to Jasmin or to its ITree semantics. We verify the back-end directly: we discuss one pass that leverages the layered semantics design to define a mixed-step semantics and enable its proof.

## 6.1 Register Renaming and Static Analyses

The Register Renaming pass renames variables to use machine register names; for example, $x := y + 3$ gets renamed to rdi := rdi + 3. This pass performs no spilling, and therefore leaves the program structure unchanged. We verify this pass in translation validation style: we use an unverified algorithm implemented in OCaml to generate a renamed program, and then a verified checker to ensure that the programs are equivalent modulo renaming.

The checker in the Register Renaming pass checks that two programs are equivalent modulo renaming. It analyzes the code to compute a *renaming*, a map from source variables to target registers, at each program point. For example, the instruction $x := y + 3$ is equivalent modulo renaming to rdi := rdi + 3 if the initial renaming (before the assignment) is $\{y \mapsto \mathsf{rdi}\}$ and the resulting one (after the assignment) is $\{x \mapsto \mathsf{rdi}, y \mapsto \bot\}$.

The correctness of the checker states that the source and target states are equal up to renaming of register variables. The simulation relation is parameterized by a renaming $\alpha : \mathsf{Var} \rightharpoonup \mathsf{Var}$, which is a partial function from variables to variables. The simulation relation for this pass states that memories should be identical, and if the renaming is defined on $x$, the variable maps coincide there:

$$\sigma \preccurlyeq_\alpha \sigma' \triangleq \sigma_m = \sigma'_m \wedge \forall x \in \mathrm{dom}\,\alpha.\, \sigma(x) = \sigma'(\alpha(x)).$$

*Key Feature: Leveraging Static Analyses.* To verify passes like Register Renaming that depend on static analyses, we develop a general approach based on simplified RHL rules. The approach entails an abstract set $\mathcal{A}$, a logical interpretation $(R_\alpha)_{\alpha\in\mathcal{A}} : \mathsf{S}\times\mathsf{S} \to \mathsf{Prop}$, a *checker* for expressions $\mathsf{chk}_e : \mathcal{A}\times\mathsf{Expr}\times\mathsf{Expr}\times\mathcal{A} \to \mathsf{Prop}$ and one for left-values $\mathsf{chk}_\ell : \mathcal{A}\times\mathsf{LVal}\times\mathsf{LVal}\times\mathcal{A} \to \mathsf{Prop}$. Informally, the abstract set collects information about the two program executions, and the interpretation maps elements of the abstract domain to a logical representation. The checkers, on the other hand, verify whether an abstract value validates a desired relation $\mathcal{S}$ (e.g., that two given expressions evaluate to the same value) and establish an updated abstract value. We identify three properties needed to incorporate checkers in the logic, namely:

$$\left.\begin{array}{r}\sigma_1\ R_\alpha\ \sigma_2\\ \mathsf{chk}_e(\alpha,e_1,e_2,\alpha')\end{array}\right\} \implies \sigma_1\ R_{\alpha'}\ \sigma_2, \qquad (\textsc{Mono})$$

$$\mathsf{chk}_e(\alpha,e_1,e_2,\alpha') \implies \models e_1 \sim e_2 : R_\alpha \Rightarrow \mathcal{S}, \qquad (\textsc{Corr}_e)$$

$$\mathsf{chk}_\ell(\alpha,\ell_1,\ell_2,\alpha') \implies \models \ell_1 \sim \ell_2 : R_\alpha / \mathcal{S} \Rightarrow R_{\alpha'}. \qquad (\textsc{Corr}_\ell)$$

The first property ensures that no information is lost after analyzing expressions (since evaluating expressions does not change the state) and allows composing the results of multiple checks. The correctness of $\mathsf{chk}_e$ ensures that evaluating $e_1$ and $e_2$ in $R_\alpha$-related states yields $\mathcal{S}$-related values. The correctness of $\mathsf{chk}_\ell$ ensures that if the check with $\alpha$ is successful and yields $\alpha'$, updating $\alpha$-related states with $\mathcal{S}$-related values yields $\alpha'$-related states.

$$\textsc{Assign}_{\mathsf{chk}}\ \frac{\mathsf{chk}_e(\alpha,e_1,e_2,\alpha') \qquad \mathsf{chk}_\ell(\alpha',\ell_1,\ell_2,\alpha'')}{\vdash \ell_1 := e_1 \sim \ell_2 := e_2 :\ R_\alpha \ \Rightarrow\ R_{\alpha''}}$$

$$\textsc{Case}_{\mathsf{chk}}\ \frac{\mathsf{chk}_e(\alpha,e_1,e_2,\alpha') \qquad \models e_1 \sim e_2 :\ R_\alpha \ \Rightarrow =\qquad \vdash c_1 \sim c_2 :\ R_{\alpha'} \ \Rightarrow\ R_{\alpha''} \qquad \vdash c_1' \sim c_2' :\ R_{\alpha'} \ \Rightarrow\ R_{\alpha''}}{\vdash \mathbf{if}\ (e_1)\ \{c_1\}\ \{c_1'\} \sim \mathbf{if}\ (e_2)\ \{c_2\}\ \{c_2'\} :\ R_\alpha \ \Rightarrow\ R_{\alpha''}}$$

$$\textsc{Case}_b\ \frac{\forall \sigma_1\sigma_2.\, \sigma_1\ \Phi\ \sigma_2 \implies (\!|e|\!)^{\mathsf{Expr}}_{\sigma_1} \approx \mathsf{Ret}(b) \qquad \vdash c_b \sim c : \Phi \Rightarrow \Psi}{\vdash \mathbf{if}\ (e)\ \{c_\top\}\ \{c_\bot\} \sim c : \Phi \Rightarrow \Psi}$$

$$\textsc{Unroll}_L\ \frac{\vdash \mathbf{if}\ (e)\ \{c\,;\mathbf{while}\ (e)\ \{c\}\}\ \{\mathbf{skip}\} \sim c' : \Phi \Rightarrow \Psi}{\vdash \mathbf{while}\ (e)\ \{c\} \sim c' : \Phi \Rightarrow \Psi}$$

**Figure 8: Specialized Relational Hoare Logic.**

The properties of checkers can be used to simplify several of the RHL rules presented in Figure 7. The first two rules of Figure 8 are simplified versions of the Assign and Case rules, whose premises rely on the checkers to establish the necessary RHL tuples. Such premises are trivially discharged in compiler transformations like Register Renaming that replace commands only when the checkers succeed. In this pass, checking two assignments $\ell_1 := e_1$ and $\ell_2 := e_2$ succeeds only if there exists a renaming $\alpha$ such that $e_2$ is a renaming of $e_1$, and similarly for the left-values.

## 6.2 Constant Branches and One-Sided Rules

The Constant Propagation pass propagates the values of constants, simplifies complex expressions, and optimizes control flow constructs if their guard is constant. For example, $i := 3\,; x := i - 2$ maps to $i := 3\,; x := 1$. This transformation optimizes control flow constructs whose guards are constant—i.e., it removes constant branches. For example, the conditional command $\mathbf{if}\ (x = x)\ \{c\}\ \{c'\}$ maps to $c$, and $\mathbf{while}\ (x < x)\ \{c\}$ maps to **skip**. We prove it correct by showing that the source and target states are equal.

*Key Feature: Specialized One-Sided Rules.* We introduce specialized one-sided rules that, for instance, deal directly with constant branches. Thus, when a while loop has a false guard, we first use the $\textsc{Unroll}_L$ rule from Figure 8, which relates a while loop to its one-step unrolling, and then the $\textsc{Case}_b$ rule to relate it to **skip**. In general, specialized one-sided rules are tailored for common compiler transformations. They pattern-match on the command on the left-hand side (i.e., the source) and set the command on the right-hand side (i.e., the target) to the transformation of the former. Conveniently, the rules have non-relational proof obligations to prove that the right-hand side correctly implements the source. Such obligations enjoy good automation in the Rocq prover, thanks to our defining a partial symbolic evaluator for commands that can reduce loop- and event-free commands to their results; for instance, it establishes that $(\!|x := 0|\!)^\star_\sigma \approx \mathsf{Ret}(\sigma[x \mapsto 0])$. All passes in Jasmin that modify the program structure use one-sided rules; further examples are Instruction Selection to introduce assignments before complex computations and Dead Code Elimination to remove useless assignments.

### 6.3 Inlining and Different Semantics

This pass replaces function calls with the body of the callee if the latter is annotated as `#inline`. The transformation is performed in the order in which functions are defined, which means that we perform inlining once per function (as its callees have already been processed). The proof for this pass is the most complex in the compiler after our changes, since a critical step in the argument requires a generic result on ITrees that is not provable in RHL directly. Proving the correctness of this pass is challenging because it requires relating an ITree with $\mathsf{Call}(f, m, v)$ events with an ITree where some of these events have already been interpreted as the semantics of the body of $f$ on a state containing $m$ and $v$. This is not expressible as one of our one-sided rules, as their equivalence holds only *after* interpreting Call events.

*Key Feature: Parameterization with Respect to Semantics.* Our solution exploits the parametricity of the logic over program semantics. Concretely, we introduce a new handler for Call events, denoted $H^I$, to bring out the effects of inlining in a new *one-step inlining semantics*. Using this new semantics, we proceed in two steps: first, we show that the semantics of the source program is equivalent to its one-step inlining semantics; and second, we use RHL to prove that the one-step inlining semantics of the source program is equivalent to the semantics of the target program.

The new handler and semantics are as follows:

$$H^I(\mathsf{Call}(f,m,v)) \triangleq \begin{cases} H^{\mathsf{c}}(\mathsf{Call}(f,m,v)) & \text{if } f \text{ is } \texttt{\#inline}, \\ \mathsf{trigger}(\mathsf{Call}(f,m,v)) & \text{otherwise}, \end{cases}$$

$$(\!|c|\!)^I_\sigma \triangleq \mathsf{interp}\Big(H^I, (\!|c|\!)^\star_\sigma\Big).$$

This handler interprets a call to an `#inline` function $f$ by executing its body using the usual call handler $H^{\mathsf{c}}$, and other calls by re-triggering the original event. Based on this handler, the one-step inlining semantics interprets calls to `#inline` functions as ITrees that correspond to their bodies instead of Call events. The bodies of `#inline` functions may themselves contain function calls, and these are not replaced, hence the name "one-step" inlining semantics.

Armed with this new semantics, we use existing lemmas from the ITree library that show that the interprocedural semantics of the source program (which interprets *every* Call event) is equivalent to interpreting all Call events *after* the one-step inlining semantics. That is, we have $\forall c, \sigma.\ \approx (\!|c|\!)_\sigma \mathsf{interp\_mrec}\big(H^{\mathsf{c}}, (\!|c|\!)^I_\sigma\big)$. Intuitively, this holds because both sides handle events the same way, but the right-hand side handles some of them "earlier" with $(\!|\cdot|\!)^I$.

Finally, we use RHL to prove that the one-step inlining semantics of the source is equivalent to the semantics of the target. That is, for every source command $c$ whose compilation is $\bar{c}$ and state $\sigma$, $\overset{u}{\approx}\ \mathsf{interp\_mrec}\big(H^{\mathsf{c}}, (\!|c|\!)^I_\sigma\big)(\!|\bar{c}|\!)_\sigma$. Critically, the left-hand side uses the one-step inlining semantics $(\!|\cdot|\!)^I$ while the right-hand side uses the call semantics $(\!|\cdot|\!)^\star$ (recall the definition of $(\!|\cdot|\!)$ in Figure 4b). We can prove this statement only because we generalize the logic and Theorem 5.3 to allow different semantics on the left- and right-hand sides.

To verify passes like Inlining that use different semantics, our Rocq formalization of the semantics and of RHL is substantially more general than what was presented in Section 5, *being parametric on the semantics of either side*. This generalization allows us to verify the front-end with a single implementation of RHL, and is essential in the proof of the Inlining pass. Besides Inlining, this generalization is needed for passes with different input and output languages (i.e., Int to Word, Reg. Array Expansion, Stack Allocation, and One Varmap).

### 6.4 Linearization and Mixed-Step Semantics

The most substantial proof in the compiler back-end is linearization. This pass transforms programs with structured control-flow into ones with jumps. For example, **while** $(e)$ $\{c\}$ transforms into **jmp** $\ell$ ; $\ell'$:$\bar{c}$ ; $\ell$:**if** $e$ **jmp** $\ell'$, where $\bar{c}$ is the compilation of $c$. The output language of linearization is Linear, an unstructured language with labels, jumps, conditional jumps, and calls. Linear states augment source states with a function name and a program counter, which are used to index the linear program. The semantics of Linear is the repeated application of a transition function on states.

*Key Feature: Mixed-Step Semantics.* An essential part of the proof of linearization entails synchronizing function calls in the source program with those in the target. With the previous big-step semantics, this was part of the (semantics) induction hypothesis. However, with the new denotational semantics, we need a means of reasoning about small-step semantics in a function-modular way: the proof of Linearization is greatly simplified when reasoning in a semantics that treats calls as uninterpreted events.

To this end, we define an auxiliary semantics, the *mixed-step semantics*, that behaves like the small-step one but triggers Call events for function calls and later interprets them analogously to the source. This allows us to prove the correctness of the Linearization pass at the uninterpreted layer of both semantics, treating function calls abstractly. Afterward, we show the equivalence between the mixed-step semantics and the small-step one as an abstract ITree equivalence between the iter and interp_mrec operators.

## 7 Related Work

We structure this section along the different lines of related work.

*Program Semantics.* There is a long line of work that formalizes semantics for nonterminating and probabilistic computations in proof assistants. Coinductive program semantics and definitions of recursive functions are mechanized in [55, 58, 61]. Semantics of probabilistic programs are mechanized in [52] (discrete setting), [3, 51] (continuous setting), and [48] (tape-based). We refer the reader to [29] for further information about (mechanized and non-mechanized) approaches to semantics of probabilistic programs.

*Verified Compilers.* There is a long history of using proof assistants for proving functional correctness and for security properties of compilers; see [19]. We now focus on three prominent examples.

First, CompCert [56] is a compiler for C, written and verified in Rocq. Early versions of CompCert used a big-step semantics and later ones a small-step coinductive semantics. All proofs are based on simulation diagrams. Secondly, CakeML [54] is a compiler for Standard ML, written and verified in HOL. The semantics is defined in a functional big-step style with clocks to deal with divergence, and proofs are based on simulation diagrams and equational reasoning. Pancake [85] is an offspring of CakeML that uses an ITree semantics, proven equivalent to the functional big-step one, to reason about the interactions between programs and the external world.

Thirdly, Vellvm [83, 86] is a compiler for the LLVM framework written and verified in Rocq. The latest version of Vellvm uses ITrees to formalize the semantics of LLVM-IR. Vellvm uses the Static Single Assignment (SSA) representation and leverages SSA-specific proof techniques for proving correctness of program transformations.

None of these compilers consider probabilistic semantics. Our approach could in principle be applied to lift their guarantees to the probabilistic setting.

*Verified Compilers for Probabilistic Languages.* ProbCompCert [75] is a formally verified compiler for a fragment of the Stan language [31]. The main component of ProbCompCert is a verified density compilation pipeline performed before compiling with the CompCert backend. Critically, this compilation pipeline is correct for a probabilistic semantics—not for the usual deterministic semantics. It would be interesting to consider whether one could enrich our relational Hoare logic with probabilistic reasoning rules in the style of probabilistic relational Hoare logic to validate this pipeline.

Zar [15] is a formally verified compiler from discrete probabilistic programs (with potentially unbounded loops) to an ITree-based representation. It uses extraction to generate samplers in OCaml and Python. It would be interesting to use our framework to generate efficient, formally verified, samplers written in assembly.

*Secure Compilation.* While the aforementioned works focus on classic compiler correctness in terms of execution traces, stronger notions have also been considered. One such notion is robust compilation [1, 63], which asks that program behaviors are preserved under adversarial contexts. A main difference between robust compilation and our work is that it allows for source and target contexts to differ, so that the latter are able to capture low-level attacks. We plan to address low-level attacks in a separate (but compatible) effort, leveraging recent developments in hardware-software contracts [49]. A different line of work studies preservation of information flow properties: Silver et al. [71] use the framework of ITrees to prove preservation of noninterference for an Imp-to-Asm compiler. Barthe et al. [21, 26, 27] prove preservation of constant-time in CompCert, a core imperative language, and an older version of Jasmin, respectively—the proof of the latter is not maintained. Constant-time (CT) has also been considered in the context of nondeterminism (e.g., proving that the Bedrock2 compiler preserves CT under nondeterminism [38]) and speculation (e.g., [12, 44, 77], which study or prove speculative CT for simplified, core compilers).

*(Relational) Programs Logics.* Relational program logics provide a general framework for reasoning about properties of two programs (i.e., 2-properties). Relational Hoare logic [34] is a program logic for reasoning about executions of two programs or two executions of the same program. Relational separation logic (RSL) [82] is a program logic for reasoning about executions of two heap-manipulating programs; it was developed concurrently with RHL. Iris [53] is a mechanized separation logic formalized in the Rocq prover, with several extensions that support relational reasoning. One of the primary applications of relational program logics is compiler correctness; for instance, Iris has been used to prove correctness of program transformations [46]. Recently, RSL has been used to prove correctness of (an idealization of) the tail-modulo constructor optimization used by the OCaml compiler [4].

Probabilistic RHL (pRHL) [28] is a relational program logic for probabilistic programs leveraging *proofs by coupling*. The logic presented in Section 5 treats random sampling abstractly, i.e., the proof shows that they are preserved *exactly*. This drastically simplifies reasoning about transformations and suffices for all passes in Jasmin. In contrast, pRHL supports sophisticated probabilistic reasoning that could be used to prove sampling-related optimizations (such as those in ProbCompCert and Zar). The logic presented in this paper could admit pRHL rules, when instantiated with the probabilistic semantics $[\![\cdot]\!]$. It would be interesting to explore such extensions to reason about optimizations [15, 57, 75] and expectations [14].

Several works develop program logics and related formalisms on top of ITrees. Vistrup et al. [79] develop a modular method for defining program logics within Iris, relying on a generic definition of weakest precondition for ITrees, alongside effect libraries that can be combined together to deal with complex languages. Silver and Zdancewic [73] formalize Dijkstra monads for weakest precondition reasoning with ITrees. Silver et al. [72] formalize a refinement calculus on top of ITrees. Finally, Michelland et al. [59] develop formally verified abstract interpreters using ITrees.

*Computer-Aided Cryptography.* There is a large body of work that formalizes cryptographic security in proof assistants [16]. In particular, the EasyCrypt proof assistant [24, 25] has been used to mechanize the security proof of many constructions, including ML-KEM [6]. EasyCrypt is based on pRHL (probabilistic relational Hoare logic) [28]. Other mechanized frameworks for cryptographic security include FCF [64], CryptHOL [30], SSProve [50], and IPDL [47]. Some works are more specifically focused on formalizing oracle systems. For instance, [76] presents a formalization of oracle systems in Lean. Their formalization has been further extended in [40] to reason about computational soundness.

Almeida et al. [8] present a general method for proving preservation of cryptographic security against adversaries with access to side-channel leakage. However, their proof is carried on pen-and-paper, at a high level of abstraction, and elides many of the important details addressed by our work. We discuss side-channels in the context of our work in the conclusion.

Low* [68] is a low-level language for writing efficient cryptographic code and verifying it. It has been used to implement cryptographic schemes, verifying for some of them a combination of functional correctness, protocol security, memory safety, and constant-time. It can be compiled to CompCert's Clight (this compilation is verified, including constant-time preservation) or to C (which is not verified). Low* is the language that implements the HACL* [66] cryptographic library, which in turn is used by the cryptographic provider EverCrypt [67]. Our work is complementary to these efforts: Theorems 2.1 and 4.1 take source-level (like those of Low* and HACL*) cryptographic guarantees down to assembly level. Combining the approaches, which would involve a Low*-to-Jasmin back-end that preserves game-based security, is interesting but further requires that source-level proofs are done in the standard computational model with a probabilistic semantics, rather than its idealized, symbolic, deterministic semantics.

Erbsen et al. [43] present the whole-system verification of a cryptographic server, including RISC-V machine code, network packets, and functional specification of X25519. It uses omnisemantics as a

unifying semantics framework [37], Bedrock2 [41] to implement, compile, and verify many software components; Fiat Cryptography [42] to generate verified, fast finite-field arithmetic; and Rupicola [65] to compile functional programs to correct Bedrock2 ones; everything is combined using Rocq as a foundation. The end result is functional correctness and safety of the entire cryptographic server: it receives network packets, authenticates them, and flips a bit in the state. Both Erbsen et al. [43] and our work aim to bring (orthogonal) guarantees down to assembly level: the focus of the former is on whole-system functional correctness, while ours is on preserving game-based probabilistic security. We discuss extending our work to system-level guarantees in Section 1.

## 8 Conclusion and Future Work

This paper lays the foundations for carrying probabilistic correctness and game-based security guarantees from source Jasmin programs to assembly programs generated by its compiler. To do so, we mechanize a framework for game-based security, introduce a new semantics for Jasmin and assembly programs, as well as a relational Hoare logic that allows applying the framework.

In independent future work, we plan to develop an assembly type system for speculative constant-time (SCT), proving that the type system enforces SCT, and that SCT entails system-level security under some standard assumptions in hardware. This effort, which builds on prior work [20, 22, 70, 74], is almost entirely independent of the current work and from the compiler proof—it only requires proving that the Jasmin compiler preserves declassification, which is very direct. However, it can be combined with the present paper to show that provable security at source level and side-channel security at assembly level guarantee system-level security against powerful attackers that can observe program leakage and have control over cache and branch predictors.

Another important direction for future work is to apply our methodology to other post-quantum cryptographic standards, including ML-DSA and SLH-DSA. There are ongoing efforts to prove functional correctness and game-based security of Jasmin implementations of ML-DSA and SLH-DSA, building on machine-checked security proofs from [17, 18]. Our framework will naturally allow these guarantees to be transferred to assembly programs, once the source level proofs are completed.

## Acknowledgments

We thank the reviewers for their time and insightful feedback. We are very grateful for the help of Yannick Zakowski and Li-yao Xia, which was instrumental in the proof of Inlining. Likewise, we are greatly thankful to Pierre-Yves Strub and Jean-Christophe Léchenet, who generously helped us with the formalization of the theorems about probability distributions and Stack Allocation, respectively. This research was supported 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; and by the *Agence Nationale de la Recherche* (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006.

## References

[1] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Catalin Hritcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 1–28. https://doi.org/10.1007/978-3-030-44914-8_1

[2] Heather Adkins and Sophie Schmieg. 2026. *Quantum frontiers may be closer than they appear.* https://blog.google/innovation-and-ai/technology/safety-security/cryptography-migration-timeline/ The Keyword (Google blog).

[3] Reynald Affeldt, Cyril Cohen, and Ayumu Saito. 2023. Semantics of Probabilistic Programs using s-Finite Kernels in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic (Eds.). ACM, 3–16. https://doi.org/10.1145/3573105.3575691

[4] Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *Proc. ACM Program. Lang.* 9, POPL (2025), 2337–2363. https://doi.org/10.1145/3704915

[5] José Bacelar Almeida, Gustavo Xavier Delerue Marinho Alves, Santiago Arranz-Olmos, Manuel Barbosa, Francisca Barros, Gilles Barthe, Lionel Blatter, Chitchanok Chuengsatiansup, Ignacio Cuevas, François Dupressoir, Luís Esquível, Benjamin Grégoire, Ruben Gonzalez, Jan Jancar, Vincent Hwang, Vincent Laporte, Jean-Christophe Léchenet, Ting han Lim, Cameron Low, Tiago Oliveira, Hugo Pacheco, Swarn Priya, Miguel Quaresma, Rolfe Schmidt, Peter Schwabe, Antoine Séré, Basavesh Ammanaghatta Shivakumar, Pierre-Yves Strub, Lucas Tabary-Maujean, Yuval Yarom, Zhiyuan Zhang, and Jieyu Zheng. 2026. *Formosa Crypto: End-to-end formally verified crypto software.* Taipei, Taiwan. https://realworldcrypto.iacr.org/2026/acceptedtalks.php

[6] José Bacelar Almeida, Santiago Arranz-Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. 2024. Formally Verifying Kyber - Episode V: Machine-Checked IND-CCA Security and Correctness of ML-KEM in EasyCrypt. In *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14921)*, Leonid Reyzin and Douglas Stebila (Eds.). Springer, 384–421. https://doi.org/10.1007/978-3-031-68379-4_12

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, and François Dupressoir. 2016. Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC. In *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9783)*, Thomas Peyrin (Ed.). Springer, 163–184. https://doi.org/10.1007/978-3-662-52993-5_9

[9] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 965–982. https://doi.org/10.1109/SP40000.2020.00028

[10] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. 2023. Formally verifying Kyber Episode IV: Implementation correctness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023, 3 (2023), 164–193. https://doi.org/10.46586/TCHES.V2023.I3.164-193

[11] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. 2009. *Verification of Sequential and Concurrent Programs.* Springer. https://doi.org/10.1007/978-1-84882-745-5

[12] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2025. Preservation of Speculative Constant-Time by Compilation. *Proc. ACM Program. Lang.* 9, POPL (2025), 1293–1325. https://doi.org/10.1145/3704880

[13] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, Vincent Laporte, and Paolo Torrini. 2026. *Artifact for "KEM-IND-CCA-Preserving Compilation of Jasmin's ML-KEM".* https://doi.org/10.5281/zenodo.21968144

[14] Martin Avanzini, Gilles Barthe, Davide Davoli, and Benjamin Grégoire. 2025. A Quantitative Probabilistic Relational Hoare Logic. *Proc. ACM Program. Lang.* 9, POPL (2025), 1167–1195. https://doi.org/10.1145/3704876

[15] Alexander Bagnall, Gordon Stewart, and Anindya Banerjee. 2023. Formally Verified Samplers from Probabilistic Programs with Loops and Conditioning. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1–24. https://doi.org/10.1145/3591220

[16] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 777–795. https://doi.org/10.1109/SP40001.2021.00008

[17] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. 2023. Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V (Lecture Notes in Computer Science)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, 358–389. https://doi.org/10.1007/978-3-031-38554-4_12

[18] Manuel Barbosa, François Dupressoir, Benjamin Grégoire, Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. 2023. Machine-Checked Security for XMSS as in RFC 8391 and SPHINCS$^+$. In *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 14085)*, Helena Handschuh and Anna Lysyanskaya (Eds.). Springer, 421–454. https://doi.org/10.1007/978-3-031-38554-4_14

[19] Gilles Barthe. 2026. Formalization of security. In *Proof Assistants and Their Applications in Mathematics and Computer Science*, Jasmin Blanchette and Assia Mahboubi (Eds.). Springer.

[20] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1267–1279. https://doi.org/10.1145/2660267.2660283

[21] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[22] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1884–1901. https://doi.org/10.1109/SP40001.2021.00046

[23] Gilles Barthe, Marion Daubignard, Bruce M. Kapron, and Yassine Lakhnech. 2010. Computational indistinguishability logic. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 375–386. https://doi.org/10.1145/1866307.1866350

[24] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2013. EasyCrypt: A Tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures (Lecture Notes in Computer Science, Vol. 8604)*, Alessandro Aldini, Javier López, and Fabio Martinelli (Eds.). Springer, 146–166. https://doi.org/10.1007/978-3-319-10082-1_6

[25] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6841)*, Phillip Rogaway (Ed.). Springer, 71–90. https://doi.org/10.1007/978-3-642-22792-9_5

[26] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 328–343. https://doi.org/10.1109/CSF.2018.00031

[27] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 462–476. https://doi.org/10.1145/3460120.3484761

[28] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 90–101. https://doi.org/10.1145/1480881.1480894

[29] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming*. Cambridge University Press. https://doi.org/10.1017/9781108770750

[30] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. 2020. CryptHOL: Game-Based Proofs in Higher-Order Logic. *J. Cryptol.* 33, 2 (2020), 494–566. https://doi.org/10.1007/S00145-019-09341-Z

[31] Guillaume Baudart, Javier Burroni, Martin Hirzel, Louis Mandel, and Avraham Shinnar. 2021. Compiling Stan to generative probabilistic languages and extension to deep probabilistic programming. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 497–510. https://doi.org/10.1145/3453483.3454058

[32] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer-Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP (2024), 789–817. https://doi.org/10.1145/3674652

[33] Mihir Bellare and Phillip Rogaway. 2006. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4004)*, Serge Vaudenay (Ed.). Springer, 409–426. https://doi.org/10.1007/11761679_25

[34] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

[35] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 353–367. https://doi.org/10.1109/EUROSP.2018.00032

[36] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 913–926. https://doi.org/10.1145/3385412.3385970

[37] Arthur Charguéraud, Adam Chlipala, Andres Erbsen, and Samuel Gruetter. 2023. Omnisemantics: Smooth Handling of Nondeterminism. *ACM Trans. Program. Lang. Syst.* 45, 1 (2023), 5:1–5:43. https://doi.org/10.1145/3579834

[38] Owen Conoly, Andres Erbsen, and Adam Chlipala. 2025. Smooth, Integrated Proofs of Cryptographic Constant Time for Nondeterministic Programs and Compilers. *Proc. ACM Program. Lang.* 9, PLDI (2025), 1692–1715. https://doi.org/10.1145/3729318

[39] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 73–87. https://doi.org/10.1109/SPW.2015.33

[40] Stefan Dziembowski, Grzegorz Fabianski, Daniele Micciancio, and Rafal Stefanski. 2025. Computationally-Sound Symbolic Cryptography in Lean. *IACR Cryptol. ePrint Arch.* (2025), 1700. https://eprint.iacr.org/2025/1700

[41] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. 2021. Integration verification across software and hardware for a simple embedded system. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 604–619. https://doi.org/10.1145/3453483.3454065

[42] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1202–1219. https://doi.org/10.1109/SP.2019.00005

[43] Andres Erbsen, Jade Philipoom, Dustin Jamner, Ashley Lin, Samuel Gruetter, Clément Pit-Claudel, and Adam Chlipala. 2024. Foundational Integration Verification of a Cryptographic Server. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1704–1729. https://doi.org/10.1145/3656446

[44] Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. 2025. Do You Even Lift? Strengthening Compiler Security Guarantees against Spectre Attacks. *Proc. ACM Program. Lang.* 9, POPL (2025), 893–922. https://doi.org/10.1145/3704867

[45] Formosa-Crypto Project. 2026. *Formosa ML-KEM: A Formally Verified Implementation of ML-KEM.* https://github.com/formosa-crypto/formosa-mlkem

[46] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498689

[47] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. 2023. A Core Calculus for Equational Proofs of Cryptographic Protocols. *Proc. ACM Program. Lang.* 7, POPL (2023), 866–892. https://doi.org/10.1145/3571223

[48] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. 2024. Asynchronous Probabilistic Couplings in

Higher-Order Separation Logic. *Proc. ACM Program. Lang.* 8, POPL (2024), 753–784. https://doi.org/10.1145/3632868

[49] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[50] Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Carmine Abate, Nikolaj Sidorenco, Catalin Hritcu, Kenji Maillard, and Bas Spitters. 2023. SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq. *ACM Trans. Program. Lang. Syst.* 45, 3 (2023), 15:1–15:61. https://doi.org/10.1145/3594735

[51] Michikazu Hirata, Yasuhiko Minamide, and Tetsuya Sato. 2023. Semantic Foundations of Higher-Order Probabilistic Programs in Isabelle/HOL. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs, Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:18. https://doi.org/10.4230/LIPICS.ITP.2023.18

[52] Joe Hurd, Annabelle McIver, and Carroll Morgan. 2004. Probabilistic Guarded Commands Mechanized in *HOL*. In *Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages, QAPL 2004, Barcelona, Spain, March 27-28, 2004 (Electronic Notes in Theoretical Computer Science, Vol. 112)*, Antonio Cerone and Alessandra Di Pierro (Eds.). Elsevier, 95–111. https://doi.org/10.1016/J.ENTCS.2004.01.021

[53] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 637–650. https://doi.org/10.1145/2676726.2676980

[54] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

[55] Xavier Leroy. 2006. Coinductive Big-Step Operational Semantics. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 54–68. https://doi.org/10.1007/11693024_5

[56] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 42–54.

[57] Jianlin Li, Eric Wang, and Yizhou Zhang. 2024. Compiling Probabilistic Programs for Variable Elimination with Information Flow. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1755–1780. https://doi.org/10.1145/3656448

[58] Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9129)*, Ralf Hinze and Janis Voigtländer (Eds.). Springer, 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

[59] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. 2024. Abstract Interpreters: A Monadic Approach to Modular Verification. *Proc. ACM Program. Lang.* 8, ICFP (2024), 28 pages. https://doi.org/10.1145/3674646

[60] Dustin Moody, Ray Perlner, Andrew Regenscheid, Angela Robinson, and David Cooper. 2024. *Transition to Post-Quantum Cryptography Standards.* NIST Interagency Report NIST IR 8547 (Initial Public Draft). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.IR.8547.ipd

[61] Keiko Nakata and Tarmo Uustalu. 2009. Trace-Based Coinductive Operational Semantics for While. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5674)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer, 375–390. https://doi.org/10.1007/978-3-642-03359-9_26

[62] National Institute of Standards and Technology. 2024. *Module-Lattice-Based Key-Encapsulation Mechanism Standard.* Federal Information Processing Standards Publication 203. National Institute of Standards and Technology, Gaithersburg, MD. https://doi.org/10.6028/NIST.FIPS.203 Published August 13, 2024. Also available as PDF: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf.

[63] Marco Patrignani, Robert Künnemann, Riad S. Wahby, and Ethan Cecchetti. 2024. Universal Composability Is Robust Compilation. *ACM Trans. Program. Lang. Syst.* 46, 4 (2024), 13:1–13:64. https://doi.org/10.1145/3698234

[64] Adam Petcher and Greg Morrisett. 2015. The Foundational Cryptography Framework. In *Principles of Security and Trust - 4th International Conference, POST 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9036)*, Riccardo Focardi and Andrew C. Myers (Eds.). Springer, 53–72. https://doi.org/10.1007/978-3-662-46666-7_4

[65] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. 2022. Relational compilation for performance-critical applications: extensible proof-producing translation of functional models into low-level code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 918–933. https://doi.org/10.1145/3519939.3523706

[66] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. 2020. HACLxN: Verified Generic SIMD Crypto (for all your favourite platforms). In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 899–918. https://doi.org/10.1145/3372297.3423352

[67] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 983–1002. https://doi.org/10.1109/SP40000.2020.00114

[68] Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Catalin Hritcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F. *Proc. ACM Program. Lang.* 1, ICFP (2017), 17:1–17:29. https://doi.org/10.1145/3110261

[69] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, Harold N. Gabow and Ronald Fagin (Eds.). ACM, 84–93. https://doi.org/10.1145/1060590.1060603

[70] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1094–1111. https://doi.org/10.1109/SP46215.2023.10179418

[71] Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. 2023. Semantics for Noninterference with Interaction Trees. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:29. https://doi.org/10.4230/LIPICS.ECOOP.2023.29

[72] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. 2023. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In *37th European Conference on Object-Oriented Programming (ECOOP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 30:1–30:26. https://doi.org/10.4230/LIPIcs.ECOOP.2023.30

[73] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434307

[74] Shixin Song, Tingzhen Dong, Kosi Nwabueze, Julian Zanders, Andres Erbsen, Adam Chlipala, and Mengjia Yan. 2025. Securing Cryptographic Software via Typed Assembly Language (Extended Version). *CoRR* abs/2509.08727 (2025). https://doi.org/10.48550/ARXIV.2509.08727 arXiv:2509.08727

[75] Joseph Tassarotti and Jean-Baptiste Tristan. 2023. Verified Density Compilation for a Probabilistic Programming Language. *Proc. ACM Program. Lang.* 7, PLDI (2023), 615–637. https://doi.org/10.1145/3591245

[76] Devon Tuma and Nicholas Hopper. 2024. VCVio: A Formally Verified Forking Lemma and Fiat-Shamir Transform, via a Flexible and Expressive Oracle Representation. *IACR Cryptol. ePrint Arch.* (2024), 1819. https://eprint.iacr.org/2024/1819

[77] Sören van der Wall and Roland Meyer. 2025. SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations. *Proc. ACM Program. Lang.* 9, POPL (2025), 1506–1535. https://doi.org/10.1145/3704887

[78] Verified-zkEVM. 2024. *VCV-io: Formalized Cryptography Proofs in Lean 4.* Verified-zkEVM. https://github.com/Verified-zkEVM/VCV-io Accessed: 2026-04-22.

[79] Max Vistrup, Michael Sammler, and Ralf Jung. 2025. Program Logics à la Carte. *Proc. ACM Program. Lang.* 9, POPL (2025), 32 pages. https://doi.org/10.1145/3704847

[80] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[81] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso

(Eds.). USENIX Association, 3655–3672. https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao
[82] Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334. https://doi.org/10.1016/J.TCS.2006.12.036
[83] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572
[84] Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. 2020. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, 71–84. https://doi.org/10.1145/3372885.3373813
[85] Junming Zhao, Alessandro Legnani, Tiana J. Tsang Ung, H. Truong, Tsun Wang Sau, Miki Tanaka, Johannes Åman Pohjola, Thomas Sewell, Rob Sison, Syeda Hira, Magnus Myreen, Michael Norrish, and Gernot Heiser. 2025. Verifying Device Drivers with Pancake. *CoRR* abs/2501.08249 (2025). https://doi.org/10.48550/ARXIV.2501.08249 arXiv:2501.08249
[86] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 427–440. https://doi.org/10.1145/2103656.2103709

## Open Science

We provide the Rocq mechanization of our framework and the modified version of the Jasmin compiler with its new proofs, which will become publicly available and open-source [13].

This is the only artifact that is needed to evaluate the paper's core contributions. It can be accessed via the URL in the references [13]. The mapping from the paper's contributions to specific definitions and theorems in the artifact is listed in the `README.md` file.

## A Background on Interaction Trees

Interaction Trees [80] provide a convenient framework to reason about interactive and nonterminating computations. Informally, ITrees represent computations as potentially infinite trees, where each non-leaf node is either a silent step or a visible event. Interaction trees are an attractive option for us because, on the one hand, unlike most operational semantics, they offer a semantics that is executable (as a functional definition) and compositional (we can reason inductively on the language syntax rather than on the more complex semantic datatype). On the other hand, unlike old-style denotational semantics, ITrees support modularity (we can deal with different types of events separately), and they allow for a comparatively simple style of coinductive reasoning, supported by a rich equational theory and relying on parameterized coinduction [84].

*Interpretation.* Interaction trees can be interpreted modularly with respect to the disjoint union of event families (written $\mathsf{E}_1 +' \mathsf{E}_2$) and the subevent relation (written $\mathsf{E}_1 \sqsubseteq \mathsf{E}_2$). The polymorphic operator $\mathsf{trigger} : \forall \mathsf{A}.\, \mathsf{E}'\ \mathsf{A} \to \mathsf{itree}\ \mathsf{E}\ \mathsf{A}$ allows lifting any subevent in $\mathsf{E}'$ to an ITree of E, given $\mathsf{E}' \sqsubseteq \mathsf{E}$. It is defined as

$$\mathsf{trigger}(e) \triangleq \mathsf{Vis}(e, \lambda x.\, \mathsf{Ret}(x)).$$

The $\mathsf{iter} : \forall \mathsf{A}\ I.\, (I \to M\ \ (I + \mathsf{A})) \to I \to M\ \mathsf{A}$ operator can be seen as a while loop in a monad $M$, repeatedly running the body (the first argument) until it produces a value of the result type A. Mappings from ITrees into a monad $M$ can be defined via *event handlers*, which are functions $H : \forall \mathsf{A}.\, \mathsf{E}\ \mathsf{A} \to M\ \mathsf{A}$ for some event subfamily E. A generic interpreter operator,

$$\mathsf{interp} : (\forall \mathsf{A}.\, \mathsf{E}\ \mathsf{A} \to M\ \mathsf{A}) \to \forall \mathsf{A}.\, \mathsf{itree}\ \mathsf{E}\ \mathsf{A} \to M\ \mathsf{A},$$

based on iter, can be used to fold a handler corecursively on an ITree. When the event handler itself is recursive, we can use the operator

$$\mathsf{interp\_mrec} : (\forall \mathsf{A}.\, \mathsf{E}'\ \mathsf{A} \to \mathsf{itree}\ (\mathsf{E}' +' \mathsf{E})\ \mathsf{A}) \to \forall \mathsf{A}.\, \mathsf{itree}\ (\mathsf{E}' +' \mathsf{E})\ \mathsf{A} \to \mathsf{itree}\ \mathsf{E}\ \mathsf{A}.$$

Since ITrees can be themselves interpretation targets, we can write modular interpreters, each for a subfamily of events, and compose them sequentially.

*Bisimulation.* A basic generalization of weak bisimilarity for ITrees, denoted $\approx$, was introduced in [80] as a coinductive-inductive relation, parametrized by a relation over results. It matches the standard notion of bisimulation for LTSs, and is equivalent to $\overset{e}{\approx}$ where the relation over events is equality.

## B Summary of Jasmin Compiler Passes

Table 2 presents a one-sentence summary of each pass in the Jasmin compiler.

## C Remaining Rules of Relational Hoare Logic

Figure 9 presents the remaining rules. The rule of consequence (or weakening), Conseq, allows us to strengthen the precondition (resp. the event postcondition) and weaken the postcondition (resp. the event precondition) of an RHL judgment.

## D Proof Effort

The Jasmin extensions presented in this work involved adapting the proofs of all passes of the compiler. This section briefly discusses the proof effort involved, highlighting the application of RHL for compiler proofs. All in all, the Jasmin framework comprises 110K lines of Rocq and 31.7K lines of OCaml code. The Rocq part includes the compiler passes, their proofs, and the proof of generic assumptions on the architectures.

The proof effort in the compiler has been reduced as a result of our work; as a rough estimate, the size of the compiler front-end proofs has decreased by 10%. Pleasingly, many preexisting proofs about expressions, straight-line code, and different well-formedness conditions extend to the new setting with negligible effort.

Table 3 shows some statistics on the proof effort before and after our work. We show the instance of the Instruction Selection pass for each supported architecture separately. The Live-Range Splitting pass is absent because it is the composition of Register Renaming and Dead Code Elimination, and therefore already accounted for. Similarly, we omit Remove Unused Results since it is implemented inside Dead Code Elimination.

**Table 2: A summary of Jasmin compiler passes.**

| Compiler pass | Summary |
|---|---|
| Preprocess | Resolve syntactic sugar and imports. |
| Int to Word | Replace word-sized integers with machine words. |
| Array Copy | Expand array copies into `for` loops. |
| Add Array Initialization | Insert array initializations at the beginning of functions. |
| Spilling | Replace spill and unspill operators with copies to and from fresh variables. |
| Inlining | Replace (some) function calls with their bodies. |
| Function Pruning | Remove unused functions. |
| Constant Propagation | Propagate the values of variables and optimize expressions with constants. |
| Dead Code Elimination | Remove unnecessary assignments. |
| Unrolling | Unroll `for` loops. |
| Live-Range Splitting | Transform the program into SSA form. |
| Remove Array Initialization | Remove array initializations. |
| Make Reference Arguments | Use intermediate pointer variables instead of array arguments. |
| Register Array Expansion | Expand register arrays into variables. |
| Remove Globals | Move local global values to the top-level. |
| Load Immediates | Store immediates in temporary registers. |
| Instruction Selection | Replace assignments with architecture-specific operations. |
| Inline Propagation | Propagate the values of inline variables. |
| SLH Instruction Selection | Replace SLH operators with architecture-specific operations. |
| Stack Allocation | Replace stack variables with concrete stack positions. |
| Lower Addresses | Use only addresses allowed by the architecture. |
| Remove Unused Results | Remove unused results of functions. |
| Register Renaming | Rename all variables to use architecture-specific registers. |
| One Varmap | Verify that local variables can be lifted to a shared state. |
| Linearization | Replace structured control flow with jumps. |
| Tunneling | Simplify chains of jumps. |
| Assembly Generation | Use concrete registers, flags, and conditions instead of variables. |

$$\textsc{Skip}\ \frac{}{\vdash \mathbf{skip} \sim \mathbf{skip} : \Phi \Rightarrow \Phi}$$

$$\textsc{Seq}\ \frac{\vdash c_1 \sim c_2 : \Phi \Rightarrow \Phi' \qquad \vdash c_1' \sim c_2' : \Phi' \Rightarrow \Phi''}{\vdash c_1 ; c_1' \sim c_2 ; c_2' : \Phi \Rightarrow \Phi''}$$

$$\textsc{Case}\ \frac{\models e_1 \sim e_2 : \Phi \Rightarrow = \qquad \vdash c_1 \sim c_2 : \Phi \wedge e_1 \Rightarrow \Psi \qquad \vdash c_1' \sim c_2' : \Phi \wedge \neg e_1 \Rightarrow \Psi}{\vdash \mathbf{if}\ (e_1)\ \{c_1\}\ \{c_1'\} \sim \mathbf{if}\ (e_2)\ \{c_2\}\ \{c_2'\} : \Phi \Rightarrow \Psi}$$

$$\textsc{Conseq}\ \frac{I_P', I_Q' \vdash c_1 \sim c_2 : \Phi' \Rightarrow \Psi' \qquad \Phi \implies \Phi' \qquad I_P' \implies I_P \qquad \Psi' \implies \Psi \qquad I_Q \implies I_Q'}{I_P, I_Q \vdash c_1 \sim c_2 : \Phi \Rightarrow \Psi}$$

**Figure 9: Extra rules of relational Hoare logic.**

**Table 3: Reduction in proof effort of the Jasmin compiler. The "Shared" column is the size—in lines of code (LOC)—of the auxiliary lemmas and definitions shared by both the previous and new proofs. The "Previous" and "New" columns present the size of preexisting and new proofs, respectively. The "Reduction" column is the reduction percentage including the "Shared" parts not modified by our work (i.e., one minus "New" plus "Shared" over "Previous" plus "Shared").**

| | LOC | | | % |
|---|---|---|---|---|
| Compiler pass | Shared | Previous | New | Reduction |
| Int to Word | 214 | 195 | 84 | 27.1 |
| Array Copy | 337 | 214 | 107 | 19.4 |
| Add Array Initialization | 352 | 263 | 137 | 20.5 |
| Spilling | 416 | 266 | 142 | 18.2 |
| Inlining | 255 | 339 | 283 | 9.4 |
| Function Pruning | 190 | 224 | 86 | 33.3 |
| Constant Propagation | 969 | 341 | 277 | 4.9 |
| Dead Code Elimination | 288 | 361 | 177 | 28.4 |
| Unrolling | 46 | 168 | 71 | 45.3 |
| Remove Array Initialization | 431 | 232 | 89 | 21.6 |
| Reference Arguments | 439 | 251 | 132 | 17.2 |
| Register Array Expansion | 674 | 243 | 219 | 2.6 |
| Remove Globals | 490 | 330 | 188 | 17.3 |
| Load Immediates | 142 | 277 | 100 | 42.2 |
| x86-64 Instruction Selection | 1681 | 254 | 90 | 8.5 |
| ARMv7 Instruction Selection | 1558 | 387 | 110 | 14.2 |
| RISC-V Instruction Selection | 583 | 173 | 72 | 13.4 |
| SLH Instruction Selection | 795 | 400 | 296 | 8.7 |
| Inline Propagation | 538 | 218 | 142 | 10.1 |
| Stack Allocation | 10 688 | 845 | 695 | 1.3 |
| Lower Addresses | 202 | 225 | 84 | 33.0 |
| Register Renaming | 433 | 445 | 198 | 28.1 |
| Total | 21 721 | 6651 | 3779 | 10.1 |