Provable Security of Cryptographic Primitives: from Algorithms to Assembly

Benjamin Grégoire

To cite this version:

Benjamin Grégoire. Provable Security of Cryptographic Primitives: from Algorithms to Assembly. Computer Science [cs]. Université Côte D'Azur, 2024. tel-04911253

HAL Id: tel-04911253
https://hal.science/tel-04911253v1

Submitted on 24 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE STIC

Provable Security of Cryptographic Primitives: from Algorithms to Assembly

---

Mémoire de synthése pour l'obtention d'une

Habilitation à Diriger les Recherches

par

Benjamin Gregoire

soutenue le 09 12 2024

HDR Jury

Rapporteurs:	Prof. Aurélien Francillon	-	EURECOM
	Prof. Peter Müller	-	ETH Zürich
	Prof. Frank Piessens	-	KU Leuven
Examinateurs:	Prof. Bruno Martin	-	Université Côte d'Azur
	Prof. Christine Paulin-Mohring	-	Université Paris Saclay

b

Résumé

Les primitives cryptographiques constituent les briques fondamentales de la sécurité informatique. Dans cette Habilitation à Diriger des Recherches, je propose des fondements théoriques et pratiques visant à établir la sécurité de ces primitives.

Dans un premier temps, je me suis concentré sur le développement de méthodes théoriques permettant de démontrer leur sécurité au niveau algorithmique. Cela a conduit à l'élaboration de la logique de Hoare relationnelle probabiliste (pRHL) et à la conception de l'assistant de preuve EasyCrypt.

Dans un second temps, mon travail s'est orienté vers l'étude des implémentations des primitives cryptographiques. Cette étape soulève de nouveaux défis, notamment en raison de la nécessité d'avoir des implémentations très efficaces. Cela incite souvent les développeurs à écrire du code en assembleur, un langage peu adapté à la vérification formelle de la correction des implémentations. Pour répondre à ce problème, nous avons introduit Jasmin, un langage dédié à l'écriture de code bas niveau hautement optimisé. Jasmin offre un haut niveau d'abstraction, ce qui simplifie la vérification formelle des implémentations et facilite le développement d'analyses statiques. De plus, afin de garantir la sécurité au niveau assembleur, le compilateur Jasmin a été formellement vérifié dans l'assistant de preuve Coq (projet Rocq).

Enfin, dans la dernière partie de mes travaux, je me suis intéressé à la résistance des implémentations face à diverses attaques par canaux cachés, telles que les attaques par cache, les attaques basées sur l'exécution spéculative (comme Spectre), et les attaques par analyse de puissance (Differential Power Analysis).

Ces travaux adoptent une approche à la fois théorique et pratique, en fournissant systématiquement des outils concrets permettant leur mise en œuvre.

Abstract

Cryptographic primitives are the fundamental building blocks of computer security. In this Habilitation to Direct Research (HDR), I present the theoretical and practical foundations required to establish the security of these primitives.

In the first part, I focused on developing theoretical methods to prove their security at the algorithmic level. This work led to the development of the probabilistic Relational Hoare Logic (pRHL) and the creation of the proof assistant EasyCrypt.

In the second part, my work shifted to studying the implementation of cryptographic primitives. This stage introduces new challenges, particularly because the need for highly efficient implementations often drives developers to write code in assembly language; a language ill-suited for formal verification of implementation correctness. To address this issue, we have introduced Jasmin, a language specifically designed for writing low-level, highly optimized code. Jasmin provides a high level of abstraction, simplifying formal verification of implementations and enabling the development of static analysis tools. Moreover, to ensure guarantees at the assembly level, the Jasmin compiler has been formally verified using the Coq proof assistant (project Rocq).

Finally, in the last part of my work, I tackled the challenge of making implementations resistant to various side-channel attacks, such as cache-based attacks, speculative execution attacks (e.g., Spectre), and power analysis attacks (Differential Power Analysis).

These contributions adopt both theoretical and practical perspectives, systematically providing tools to facilitate their implementation.

Acknowledgments

Comme on me l'a souvent fait remarquer, j'aurais dû écrire cette Habilitation à Diriger des Recherches il y a des années. Cela aurait réduit le risque d'oublier de remercier quelqu'un.

Je tiens à remercier les rapporteurs et les membres du jury, qui ont accepté très rapidement d'évaluer ce travail et ont su se libérer pour permettre la soutenance de cette HDR.

Je remercie toute l'équipe Marelle/Stamp qui m'a chaleureusement accueilli pendant de nombreuses années. J'ai particulièrement apprécié travailler avec Laurent. Même si c'est moins le cas aujourd'hui, j'ai bon espoir que nous aurons de nouvelles occasions de collaborer à nouveau. Je tiens aussi à remercier Enrico, avec qui nous avons écrit quelques articles, beaucoup discuté et également grimpé. Une dédicace spéciale à Yves, qui m'a toujours encouragé à écrire cette HDR. J'espère que tu ne m'en voudras pas trop de l'avoir rédigée maintenant que j'ai rejoint l'équipe SPLiTS.

Il est clairement impossible de citer toutes les personnes qui m'ont encouragé à écrire cette HDR, mais je tiens au moins à mentionner Nataliia, qui, je ne sais pourquoi, a su trouver les mots pour me convaincre lors de la remise des prix de l'université de Nice, où elle et Swarn ont reçu une distinction.

Proche de moi, il y a aussi ma nouvelle équipe SPLiTS avec Illaria, Manuel, Tamara, Davide et Martin (qui n'est pas officiellement dans l'équipe, mais on fait comme si). Je tiens à tous les remercier. Je trouve très excitant de commencer cette nouvelle aventure avec vous, et je suis sûr qu'elle sera fructueuse. Ces derniers temps, j'ai particulièrement travaillé avec Martin sur de nouvelles versions des différentes logiques d'EasyCrypt, ce qui a permis d'établir des résultats de complétude très intéressants (je sais, Gilles, je vieillis).

Je dois remercier Tamara, qui a énormément contribué à la rédaction de cette HDR en fournissant motivation, encouragements et de nombreuses relectures. De plus, j'apprécie ton écoute et ton attention.

Merci à Christine et Nathalie, sans qui mon travail serait un calvaire. Votre abnégation est incroyable. Je pense qu'il n'est pas toujours facile de travailler avec quelqu'un capable de réserver un billet d'avion pour le mois de juillet alors qu'il doit partir en juin. Vous avez toujours su me soutenir, me décharger et m'aider dans la

bonne humeur, malgré toutes mes incompétences. Parfois même dans des domaines inattendus : je me souviens encore de Nathalie faisant du lobbying au sein de l'Inria pour que je sois recruté.

Je remercie également tous les ingénieurs avec qui j'ai eu la chance de travailler. Je pense que votre travail est essentiel et devrait occuper une place beaucoup plus importante au sein de l'institut. Commençons par Anne, qui a lancé le développement d'EasyCrypt, puis Maxime, avec qui les discussions ont toujours été intéressantes et éclairantes. Plus récemment, j'ai collaboré avec l'équipe du SED, composée de Thibaut, Romain, Côme et Jean-Christophe. J'apprécie particulièrement l'énergie que vous apportez aux projets ainsi que les questions pertinentes que vous soulevez.

Je tiens également à remercier les différents étudiants que j'ai encadrés ou avec qui j'ai collaboré : Assia, Maxime, Santiago, César, Fernando, Jorge-Luis, Michael, Julien, Sylvain, Cécile, Swarn, Basavesh, Santiago (le second) et Lucas.

Naturellement, il y a aussi de nombreux chercheurs à remercier. Commençons par la période EasyCrypt/Zoocrypt, avec Benedict, ainsi que François et Pierre-Yves, avec qui nous avons peut-être bu un peu trop de mojitos en Espagne – je crains que le code d'EasyCrypt porte encore les stigmates de cette période! La collaboration ne s'est pas limitée à EasyCrypt : il y a eu également tous nos travaux sur le masking, auxquels Sonia et Pierre-Alain ont également contribué. À chaque fois, ces projets ont été réalisés dans la bonne humeur et avec beaucoup d'enthousiasme.

En Espagne, il y avait aussi Vincent, qui est ensuite venu à Sophia pour une année. Cela a marqué le lancement de Jasmin et des travaux sur la préservation du constant-time. Il n'est pas toujours facile d'être compris par les autres, surtout pour un dyslexique capable d'écrire "b" tout en prononçant α, mais Vincent possède cette incroyable capacité de traduction automatique.

Actuellement, il y a tout le groupe Formosa, avec Andreas, Bacelar, Lionel, Manuel, Peter, Tiago, et beaucoup d'autres déjà cités plus haut. Ce groupe est une formidable source de motivation et d'inspiration. C'est incroyable d'avoir la chance de travailler avec un groupe aussi large de personnes exceptionnelles, qui ont chacune des compétences différentes et qui permettent d'avancer ensemble dans une direction commune.

Il suffit de regarder ma liste de publications pour constater que Gilles occupe une place plus qu'importante dans ma carrière. Il a commencé par être membre de mon jury de thèse, et il en a profité pour faire des blagues à ma grand-mère. Il a permis mon recrutement en tant que chercheur et a su croire en moi pour ce qu'il voyait, et non en fonction de mes diplômes – chose remarquable dans le pays des grandes écoles.

Gilles, tu es un chercheur exceptionnel, une source d'inspiration, de motivation et de bonne humeur. Tu m'as toujours soutenu, notamment dans les moments où ma motivation pour le travail faiblissait. Merci pour tout.

En écrivant ces remerciements, je prends conscience de l'importance que l'amitié a pour moi. Merci à vous tous pour cela.

Finalement, je tiens à remercier ma compagne Nathalie et mes enfants, Jasmin et Gaspard. J'espère que mes enfants réaliseront bientôt que leur père n'est pas un super-héros, même si le super-héros est nain dans Le Seigneur des Anneaux. Vous arrivez à l'âge de prendre votre envol : croyez en vous !

Ma p'tite Nat, merci pour ta patience, ta compréhension et la liberté que tu me laisses. Quand on est jeune, on rêve de coups de foudre et d'amours passionnés. Après bientôt 20 ans de mariage, je réalise à quel point un amour qui grandit et se consolide jour après jour est plus magnifique encore. Je te dois tous nos succès.

Trente-un ans après le décès de mon père, il me manque toujours autant. Merci, Papa, pour tout ce que tu as su m'apporter dans cette période trop courte.

Contents

1 Introduction 1

2 Formal proofs of cryptographic primitives 3
2.1 Formal method to reason about cryptographic proofs 4
2.2 Main contributions 14

3 Cryptographic implementations 15
3.1 Problem with cryptographic implementation 15
3.2 Proposed solutions 17
3.2.1 Efficiency 17
3.2.2 The certified Jasmin compiler 18
3.2.3 Type checking and safety analysis 21
3.2.4 Proving Jasmin programs 24
3.3 Main contribution 27

4 Side Channels 29
4.1 Problem with side channel attacks 29
4.2 Preservation of Constant Time 30
4.2.1 Establishing Constant time 33
4.3 Speculative Constant Time 34
4.4 Masking 39
4.5 Main contribution 43

5 Perspectives 45

Chapter 1

Introduction

I defended my PhD entitled *Compilation de termes de preuves : un (nouveau) mariage entre Coq et OCaml* in December 2003. My thesis work established the theoretical and practical foundations of computational proofs in the Coq tool, and was also an important step towards certified compilers. This work [89, vm_compute] was a major element in the evolution of Coq by providing the infrastructure for reflexive proofs or computational proofs, which are now widely used. After that, I did a post-doc at Inria Sophia Antipolis, where I worked on the proof of the C compiler : CompCert, with Yves Bertot. I also took advantage of this period to continue my work on reflexive proofs. My work in this area is founded on the methodology I developed, with Assia Mahboubi, for the ring tactic [87, Ring/Field]: decision of ring equalities. This work has been extended to field equalities and comparison in linear and non-linear arithmetic by Laurent Théry and Frédéric Besson, respectively. Those tactics are used extensively in Coq today. With Laurent, I have also developed a library for large integers [78, bignums], for primality proofs [78, 79, coqprime] and in geometry [70].

With Maxime Dénes, I've continued to improve Coq's computational capacity by developing compilation to assembler [59], but also by integrating in Coq two essential components for efficient programs: machine integers and arrays [63]. The addition of primitive integer and array has enabled the integration in Coq of proofs generated by the SAT solvers [63] and SMT solvers [58]. This work initiated the integration of automatic provers into Coq (now taken over by Chantal Keller).

But this work was carried out after I joined Gilles Barthe's team at Inria Sophia-Antipolis in 2004 for a post-doc. I was recruited as a researcher in the Everest team in September 2005, shortly after the birth of my first son Jasmin. It was the start of a particularly fruitful collaboration, that continues to this day, and I hope, that will continue for a long time to come.

It was during this post-doc that I began to study much more theoretical aspects of Coq's foundations: type theory. In particular with type systems for the termination

of recursive functions [86, 81, 74, 72, 65], type erasure in conversion [85] and improved dependent filtering [75] in the Calculus of Inductive Constructions.

In 2007, I began to focus on the proof of probabilistic programs, and more specifically on proofs of cryptographic primitives. This was during Santiago Zanella-Béguelin's thesis. I started at a relatively abstract level: that of algorithms. This work evolved more and more towards concrete proofs of implementation. Fifteen years later we are now able to provide very strong guarantees at assembly level and even at hardware level.

This *mémoire d'habilitation* summarises those 16 years of research in the topic of provable security of cryptographic primitives from 2007 to 2023. It focuses on three complementary topics: security proofs, functional correctness proofs, and protection against side-channel attacks.

The complete list of my publications since my PhD is listed at the end of the summary. In the text, citations to my own work between 2007 and 2023 appear in plain style as in [68] whereas citations to other works appear in alpha style as in [AP13].

During my research I have always attached great importance to providing new theoretical results but also, and I'm particularly proud of this, to developing tools that enable these results to be put into practice. This has given rise to different tools, some of which are more exploratory research prototypes and some others which are much more perennial. They are cited after the bibliography in this style: [EasyCrypt].

As you can see in the publication list, all my publications are done with co-authors, sometimes with many co-authors. It is always hard to say what is the exact contribution of each authors in a publication or in a development. Hence, I have decided to use *we* everywhere in this *mémoire d'habilitation*.

Chapter 2

Formal proofs of cryptographic primitives

Cryptography plays a key role in the security of modern communication and computer infrastructures; therefore, it is of paramount importance to design cryptographic systems that provide strong security guarantees. To achieve this goal, cryptographic systems are supported by security proofs that establish an upper bound on the probability that a resource-constrained adversary could break the system. In most cases, security proofs are reductionist; that is, they construct from an (arbitrary but computationally bounded) adversary that would break the security of the cryptographic construction with some reasonable probability, another computationally bounded adversary that would break a hardness assumption with a similar probability.

This approach, known as provable security, is theoretically capable of delivering rigorous and detailed mathematical proofs. However, new cryptographic designs (and consequently their security analyses) are becoming increasingly complex. There is a growing emphasis on shifting from algorithmic descriptions to implementation-level descriptions that account for practical details, such as recommendations from standards (where available) and the potential impact of side channels. As a result, cryptographic proofs are becoming increasingly error-prone and difficult to verify.

In this chapter, we present our efforts to address these concerns through the development of machine-checked frameworks that support the construction and automated verification of cryptographic systems. Figure 2.1 provides an overview of my research in this area.

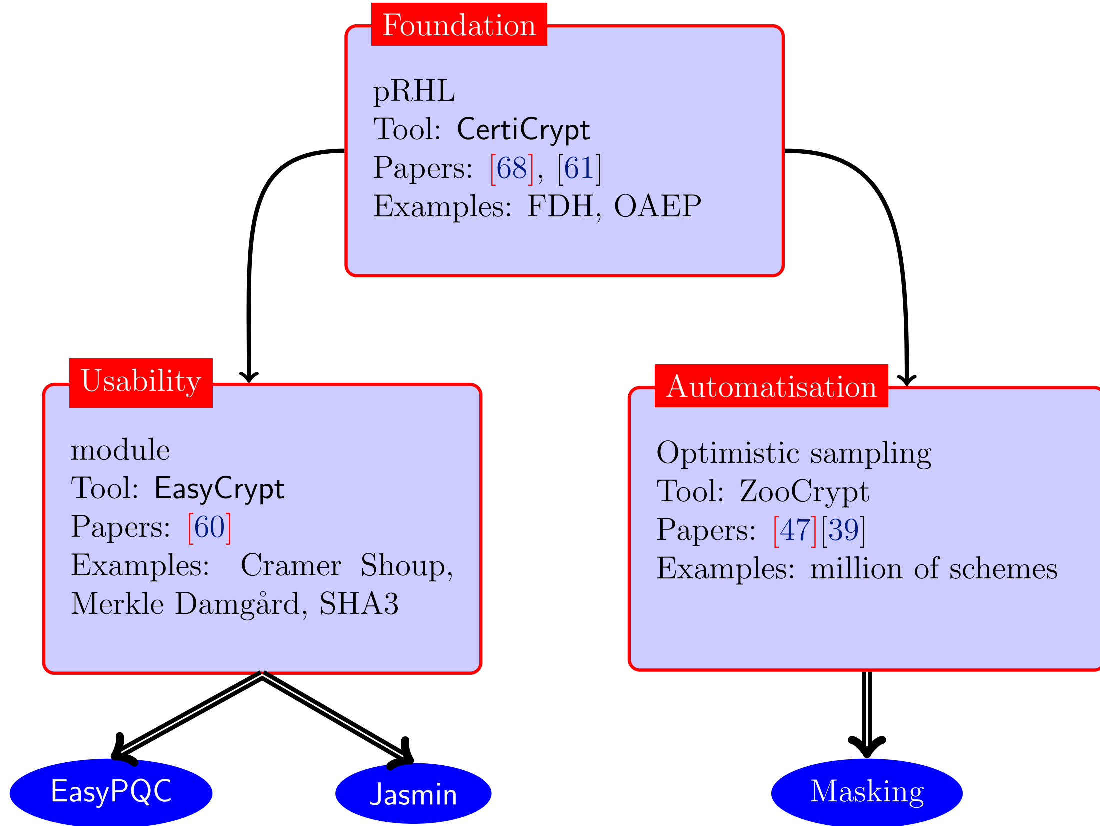


Figure 2.1: Organisation of the research

2.1 Formal method to reason about cryptographic proofs

In 2007, we began exploring methods for formalizing proofs of cryptographic primitives in Coq. Cryptographers felt that their field was facing a crisis of rigor and were seeking a tool to address this issue. The approach that seemed most reasonable to us was the game-playing technique [BR06], or proofs by reductions.

From a formalization perspective, the main advantage of this approach is that everything can be expressed in terms of probabilistic programs:

- Cryptographic schemes are represented as probabilistic programs;
- Attackers are higher-order (second-order) probabilistic programs, parameterized by oracles (procedures). These oracles can be, for example, a random oracle, a *decryption* oracle for IND-CCA, or a *sign* oracle for EUF-CMA;

- Security notions are expressed in the form of a game between an attacker and the cryptographic scheme. Again, this is a program that represents the interactions between the cryptographic scheme and the adversary, making security notions third-order programs.

The proofs in this model involve bounding the advantage of the adversary, i.e., their winning probability:

$$\Pr[\mathsf{EUF\text{-}CMA}_S^A : W] \leq \epsilon$$

where EUF-CMA represents existential unforgeability under a chosen message attack game, A is the adversary, S is the signature scheme, W is the winning event, and ϵ is the exact security bound.

In this game, the adversary's goal is to generate a valid signature without knowing the secret key used in the signature. To achieve this, the adversary has access to the *sign* oracle (which depends on the secret key) and to random oracles when the proof is conducted in the random oracle model (ROM).

To establish this type of inequality, a sequence of transformation steps is applied to the games. We can distinguish two main categories of transformations:

- **Bridging step** The game and the event are transformed without changing the winning probability:

$$\Pr[G_1 : E_1] = Pr[G_2 : E_2]$$

- **Splitting step** The probability is split into two parts:

$$\Pr[G : E] \leq \Pr[G_1 : E_1] + Pr[G_2 : E_2]$$

  An example of this kind of step is the *upto-bad* step, where we have to prove that G and G_1 are equivalent up to the point where a bad event is triggered. In this case, we need to bound both the winning probability in G_1 and the probability that the bad event occurs.

After a sequence of transformations, we need to *close* the proof. This can be done in two ways: either by directly calculating the probability of the event (for example, when the event is *a fresh random value equals a particular constant*), or by showing that the current game is an instance of a cryptographic assumption, such as the Computational Diffie-Hellman (CDH) problem. The latter case is called a reduction because it reduces the original problem to a specific instance of a cryptographic assumption.

Formalizing this type of proof requires the ability to reason about probabilistic programs and their semantics. We also need methods for proving program equivalence exactly (bridging step) or approximately (upto-bad). Finally, we need a method to bound the probability of an event occurring in a game.

CertiCrypt: In 2009, we published the paper [68] and introduced the Coq library: CertiCrypt [CertiCrypt]. This work presents the theoretical foundations for proving cryptographic primitives. A key contribution of this work is the introduction of probabilistic relational Hoare logic (pRHL), a logic that allows proving the equivalence of programs. This logic forms the foundation of CertiCrypt and all the work presented in this chapter (see Figure 2.1).

The pRHL logic was formally verified in Coq. To achieve this, we developed a library in Coq that defines the semantics of pWhile, a simple while-language with procedure calls and random sampling.

A pRHL judgment has the following form:

$$\{\phi\}\ G_1 \sim G_2\ \{\psi\}$$

where G_1 and G_2 are probabilistic programs, ϕ is the pre-condition, and ψ is the post-condition. The pre- and post-conditions are relations over memories (the state of the programs). The logic allows for relating the execution of two probabilistic programs.

The interpretation of a traditional Hoare judgment over a non-probabilistic program is typically defined as follows:

$$\{\phi\}\ G\ \{\psi\} ::= \forall m,\ \phi\ m \Rightarrow \psi\ [\![G]\!]_m$$

In other words, for all initial memories m satisfying the pre-condition, the final memory $[\![G]\!]_m$ (the memory obtained after evaluating G starting from m) should satisfy the post-condition. The same idea applies to the interpretation of a pRHL judgment, except that the pre- and post-conditions are relations over memories rather than predicates, and the semantics of a probabilistic program is a distribution of memories rather than a single memory. Since the post-condition is a relation over memories, we need a method to lift it to a relation over distributions.

An appropriate method to do this is probabilistic coupling [Lin02]. Let d_1 be a distribution over A and d_2 be a distribution over B and ψ a relation over $A \times B$, the lifting of ψ to a relation over $\mathrm{Distr}(A) \times \mathrm{Distr}(B)$ is defined by

$$d_1 \Downarrow^{\psi} d_2 ::= \exists\ d : \mathrm{Distr}(A \times B), \begin{cases} \pi_1(d) = d_1 \\ \pi_2(d) = d_2 \\ \forall a\ b, (a, b) \in d \Rightarrow a\ \psi\ b \end{cases}$$

d is a distribution of $A \times B$ statisfying three properties. The first condition ensures that the projection of d onto its first component follows the same distribution as d_1. The second condition ensures that the projection onto the second component follows the same distribution as d_2. The last condition ensures that all pairs (a, b) in d (i.e., those with non-zero probability) satisfy ψ.

Using this definition, the interpretation of a pRHL judgment is:

$$\{\phi\}\ G_1 \sim G_2\ \{\psi\} ::= \forall\ m_1\ m_2,\ m_1\ \phi\ m_2 \Rightarrow [\![G_1]\!]_{m_1} \Downarrow^{\psi} [\![G_2]\!]_{m_2}$$

Recall that since G_1 and G_2 are probabilistic programs, their denotations $[\![G_1]\!]m_1$ and $[\![G_2]\!]m_2$ are distributions over memories.

It is important to understand the consequences of establishing such a judgment with respect to probabilities. Assume we can prove $\{\phi\}\ G_1 \sim G_2\ \{E\langle 1\rangle \Rightarrow F\langle 2\rangle\}$ where the notation $E\langle 1\rangle$ (resp. $F\langle 2\rangle$) means that the predicate E is evaluated in the first memory (resp. second memory). Then, for all memories m_1 and m_2 such that $m_1\ \phi\ m_2$, we have $\Pr[G_1, m_1 : E] \leq \Pr[G_2, m_2 : F]$. This follows form the interpretation of the judgment and form monotonisity of the expectation:

$$\Pr[G_1, m_1 : E] = \Pr[d : E\langle 1\rangle] \leq \Pr[d : E\langle 2\rangle] = \Pr[G_2, m_2 : F]$$

Similarly, the judgment $\{\phi\}\ G_1 \sim G_2\ \{E\langle 1\rangle \Leftrightarrow F\langle 2\rangle\}$ allows us to conclude that $\Pr[G_1 : E] = \Pr[G_2 : F]$.

While it is possible to define a syntactic criterion to decide if two programs are equivalent up to a bad event, it is also possible to define it using the logic. For example:

$$\mathrm{bad}\langle 1\rangle \Leftrightarrow \mathrm{bad}\langle 2\rangle \wedge (\neg\mathrm{bad}\langle 1\rangle \Rightarrow E\langle 1\rangle \Leftrightarrow F\langle 2\rangle)$$

is a suffisiant post-condition to prove that

$$|\Pr[G_1 : E] - \Pr[G_2 : F]| \leq Pr[G_i : \mathrm{bad}]$$

While a syntactic criterion is very convenient from a proof/automation point of view, this logical criterion provides much more expressivity in defining bad events.

I believe a key feature of the logic is that pre- and post-conditions are relations over memories, rather than distributions. While the judgment allows us to relate the probabilities of events in two programs, most of the reasoning steps do not involve distributions. Naturally, distributions arise from time to time, particularly in the rule relating two random samplings. This rule provides sufficient conditions to demonstrate the existence of a coupling.

Adversaries In cryptography, security properties are universally quantified over adversaries. In CertiCrypt/EasyCrypt, adversaries are represented by *abstract* programs, i.e., programs for which the code is unknown. From the perspective of the logic, this has significant implications: the logic needs to provide rules to reason about unknown code. Without restrictions on the unknown code, this would be impossible. The typical restriction on adversaries in cryptographic proofs is usually on their complexity; specifically, adversaries are assumed to be probabilistic

polynomial-time (PPT). While this is important for cryptographic assumptions, it is irrelevant from the logic's point of view. For the logic, the critical information is which parts of the memory can be read and written by the adversary. Reading is important because it influences the adversary's behavior, while writing is important because it affects the behavior of other parts of the program.

To provide intuition on how we can have logical rules for adversaries, let's start with Hoare logic. Since the adversary can call oracles, we can view it as a loop that executes its code and, from time to time, executes the code of the oracles. Similar to logical rules for while loops, a natural idea is to prove that the oracle preserves an invariant, i.e., if the invariant is true at the beginning of the oracle's execution, it remains true at the end. Since the adversary executes some code between two calls to oracles, it is important to show that it cannot break the invariant. A sufficient condition to ensure this is to make sure that the invariant does not depend on the part of the memory that can be modified by the adversary, or, symmetrically, that the adversary cannot write to the part of memory read by the invariant.

For relational logic, a trivial rule is that if the two programs are identical and the pre-condition ensures the equality of the state (at least the parts of the state that the program depends on), then the final states will be equal (at least the parts that have been written). The advantage of this rule is that it is independent of the code of the program. The disadvantage is that it requires exactly the same code on both sides, so it does not allow modifications to the oracles passed to the adversary. Our solution is to blend this trivial approach with the Hoare logic rule presented earlier.

The rule for adversaries starts by requiring that both sides involve calls to the same adversary (i.e., the same abstract program), although the oracles on both sides can differ. The precondition ensures that the part of the memory readable by the adversary is the same on both sides and that the relational invariant is initially true. It ensures that the part of the memory writable by the adversary will be equal after its execution and that the invariant remains valid. The rule requires that the relational invariant is preserved by oracle pairs, and assuming equality of oracle arguments, that the results will be equal. This allows maintaining the equality of the adversary's state on both sides. The rule can be proved by simple induction on the adversary's code, using the rules of the logic.

A more intricate rule is provided for equivalence *up to-bad* (with adversary). This rule ensures that the relational invariant is preserved up to the point where a bad event is triggered, that the bad event remains stable once it is set, and that the code terminates with probability one. This last condition is required by the interpretation of the logic, as the notion of coupling implies that the probability of termination, i.e of true, is equal in both distributions.

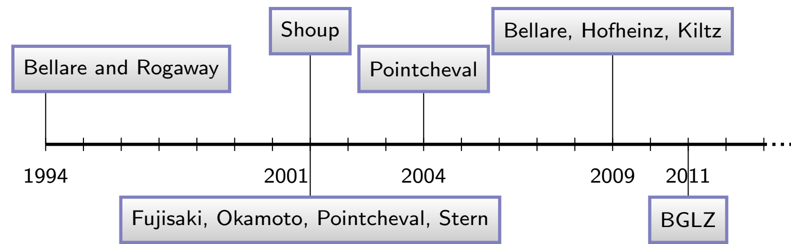


Figure 2.2: OAEP history

OAEP One of the major achievements of CertiCrypt is the proof of IND-CCA semantic security for Optimal Asymmetric Encryption Padding (OAEP) [61]. This is because this proof had a history of failures and patches (see Figure 2.2) and the fact of having a machine-checked proof convinced the cryptographers that our tool had the sufficient maturity level to succeed in providing proofs that cryptographers had difficulty doing by hand.

The history of OAEP security is fraught with difficulties. The original 1994 paper by Bellare and Rogaway [BKR94] proved that, under the assumption that the underlying trapdoor permutation family is one-way, OAEP is semantically secure against chosen-ciphertext attacks. However, in 2001, Shoup [Sho01] discovered that this proof only established security against non-adaptive chosen-ciphertext attacks, rather than the stronger version of IND-CCA, which allows an adversary to adaptively obtain the decryption of ciphertexts of its choice. Shoup proposed a modified scheme, OAEP+, which is secure against adaptive attacks under the one-wayness of the underlying permutation, and provided a proof of the adaptive IND-CCA security of the original scheme when used with RSA with public exponent $e = 3$.

At the same time, Fujisaki, Okamoto, Pointcheval, and Stern [FOPS04] proved that the original formulation of OAEP is secure against adaptive attacks under the assumption that the underlying permutation family is partial-domain one-way. Since, in the particular case of RSA, this assumption is no stronger than (full-domain) one-wayness, this result established the adaptive IND-CCA security of RSA-OAEP. In 2004, Pointcheval [Poi05] provided a different proof of the same result, filling several gaps in the reduction from [FOPS04], which led to a weaker bound than originally stated. Nonetheless, the inaccurate bound from [FOPS04] has remained the reference bound in practical analyses of OAEP, as noted in [Bol09].

Finally, Bellare, Hofheinz, and Kiltz [BHK09, BHK15] pointed out ambiguities in the definition of IND-CCA, leading to four possible formulations (all used in the literature), and questioned which definition is employed in the statements and proofs concerning OAEP.

Achiving this proof in CertiCrypt was a challenging task. One key point was to

devellope method to capture the informal notion used by cryptographer: “the value is uniformly distributed and independent from the adversary’s view”. To capture this notion, we have devellopped a logic for Eager/Lazy sampling [64]. The logic allows interprocedural code motion, in which sampling statements are moved from an oracle to the main command of the game or, conversely, from the main command to an oracle.

EasyCrypt The paper on OAEP [61] is both an achievement and an admission of failure. It was an achievement because it was the first important proof performed using tools like this.

It was a failure because it became clear that the amount of work and expertise in formal proofs required to produce this type of proof using CertiCrypt was a major obstacle that would prevent cryptographers from using our tools. So we had to come up with something else. This led us to develop EasyCrypt [EasyCrypt], which we are still developing today. The idea was to develop a prover independent of Coq, completely dedicated to the proof of probabilistic programs.

There were 4 main difficulties in developing the tool within Coq:

- Error messages: When building a tool on top of Coq, it is hard to provide good error messages with proper localization. This is relatively simple if the tool is implemented in a language like OCaml.
- Reduction rules and differences between Prop and bool: Coq distinguishes between the set of Propositions (Prop) and the set of decidable propositions (bool). For a non-expert, this is very surprising. Furthermore, most predicates in Coq do not reduce, so $\mathsf{True} \wedge P$ is not convertible with P—they are only provably equivalent. This is the case for the boolean version true && b but not for b && true. While it is possible to provide a good automatic Coq tactic to solve this (as SSReflect shows), it creates significant overhead.
- Lack of integration with SAT/SMT solvers: The proof obligations obtained using pRHL are very different from what you usually obtain in Coq. They are large, but most can be trivially proven. Thus, it is important to be able to solve them mostly automatically.
- Proved logical rules versus usable rules: While it is nice to have a sound and possibly complete Hoare logic, this doesn’t make it usable. Moving from a sound logic to a user-friendly one represents a lot of work. Coq is very useful for proving that the logic is correct, but it is less convenient for making it user-friendly.

After 10 years of development, I am not completely sure it was the right choice, in the sense that the trusted base of EasyCrypt is much larger than that of CertiCrypt.

However, we are still developing and using EasyCrypt, and I'm not sure that would have been the case if we had continued with CertiCrypt.

The initial idea of EasyCrypt was to have a strong connection with SMT provers. The objective was to discharge most of the side conditions of the logic rules using SMT. Furthermore, the idea was to derandomize the programs, i.e., to split programs into two parts. The first part would start by sampling the randomness, while the second part was deterministic. Naturally, this derandomization is done through automatic program transformation. Thus, a weakest precondition calculus can be used to compute the precondition of the deterministic part. Finally, the relational logic can be used to build the coupling between the random parts of each program. This is the high-level pitch. In practice, things are more complex because derandomizing a loop is not always possible, and the same challenge arises with adversaries.

The second idea was to use a module system to structure games. In that setting, adversaries are just abstract functors. Giving oracles to adversaries corresponds to functor application, and performing a cryptographic reduction simply corresponds to functor abstraction and application.

The result of this work was published in [60] and received the Best Paper Award. I think this was a very important step. First of all, it was our first publication in a cryptography conference—previous publications had been either in programming language or security conferences. But above all, we were able to convince cryptographers that our work could solve their rigor problem and make a meaningful contribution to their field.

Zoocrypt [ZooCrypt]: The CertiCrypt and EasyCrypt languages are very expressive; there are no real restrictions on the programs and security properties that can be expressed. The only limitation is that they apply to probabilistic programs but not to quantum programs. The downside is the degree of automation we can provide.

The paper [47] therefore follows a different approach: we consider a very restricted language and security properties. Programs are restricted to encryption schemes that can be described using bitstrings, concatenation, exclusive-or, random sampling, hash functions (random oracle), and trapdoor permutations. Security properties are limited to IND-CPA and IND-CCA2. The upside is a fully automatic proof strategy based on a simple logic.

The judgments for the IND-CPA logic have the following form:

$$\models_p c : \phi$$

Where event ϕ can be Guess, $\mathrm{Ask}(H, e)$, or a conjunction of events. Guess corresponds to the adversary correctly guessing the hidden bit b in the CPA game, and

Ask(H, e) corresponds to the adversary querying the random oracle $H(e)$. p is the winning probability of the adversary.

The objective here is not to provide a full description of the logic; interested readers can refer to the paper directly. What I would like to emphasize is a simple rule called *optimistic sampling*. The rule is as follows:

$$\frac{\models_p c : \phi \quad r \notin \mathcal{R}(e)}{\models_p c\{e \oplus r/r\} : \phi\{e \oplus r/r\}}$$

Here, r is a random value, e is an expression in the language, and $\mathcal{R}(e)$ is the set of random values used in e. The validity of the rule follows from the fact that r and $e \oplus r$ follow the same distribution.

Assume that we would like to prove $\models_p c^* : \phi^*$ using the rule. The difficulty is to find r, e, c, and ϕ such that $c^* = ce \oplus r/r$ and $\phi^* = \phi e \oplus r/r$ with $r \notin \mathcal{R}(e)$. One way to achieve this is to find two contexts, C and Φ, along with r and e, such that $r \notin \mathcal{R}(C) \cup \mathcal{R}(\Phi)$, $c^* = C[e \oplus r]$, $\phi^* = \Phi[e \oplus r]$, and $r \notin \mathcal{R}(e)$. In this case, we can take $c = C[r]$ and $\phi = \Phi[r]$.

While relatively simple, this rule is key to our work on masking, presented in Chapter 4. Being able to find these contexts efficiently was not critical because the expressions we manipulated were relatively small. However, efficiency is crucial for the work in Chapter 4.

The paper also proposes fully automated methods for finding attacks against chosen-plaintext and chosen-ciphertext security. Our methods are inspired by static equivalence [AF01] and exploit the algebraic properties of trapdoor permutations to find attacks against realizations of schemes that are consistent with computational assumptions. We demonstrated the strengths of our methods by implementing a toolset for fully automatic analysis of a set of user-given or machine-generated schemes. We generated more than one million examples and used the toolset to analyze their (in)security. The output of the analysis is a database that records, for each scheme and set of assumptions, either an adversary that breaks the scheme's security or a formal derivation that proves its security.

The smallest scheme found through this method is ZAEP: our tool was able to automatically prove that it is IND-CPA but not that it is IND-CCA. Using stronger hypotheses on the trapdoor permutation, the scheme was proven IND-CCA in [BPB12].

EasyPQC: The area of post-quantum cryptography (PQC) focuses on classical cryptosystems that are provably secure against quantum adversaries. PQC is based on computational problems that are conjectured to be hard for quantum computers, e.g., the learning with errors problem [Reg05]. Simply relying on such assumptions, however, is insufficient to ensure security against quantum attackers; one must also verify that a security reduction holds in the quantum setting.

A natural question to ask is whether we need a fundamentally different approach to the design of formal verification tools to capture these results, which seem tantalizingly close to the classical setting. For example, Unruh [Unr19] suggests that EasyCrypt is not sound for quantum adversaries. Concretely, [Unr19] claims that the CHSH protocol, which is secure in the classical setting but not in the quantum setting, can be proved secure in EasyCrypt. While this point is moot because the EasyCrypt logics were not designed (or claimed) to be sound for the quantum setting, it does raise an important question that we address in the paper [12]:

1. Can we adapt the EasyCrypt program logic and libraries in a way that guarantees their soundness for PQC proofs?

2. Is the resulting framework expressive and practical to use?

In the paper, we affirmatively answer both questions. For the first, we provide a post-quantum relational Hoare logic (pqRHL), a mild variant of EasyCrypt's probabilistic relational Hoare logic (pRHL), and prove that pqRHL is sound for reasoning about quantum adversaries.

To answer the second question, we have developed a new implementation of EasyCrypt for verifying post-quantum security proofs. The advantage of the new version is its compatibility with the original EasyCrypt. We have used the new tool to provide mechanized proofs of PRF-based MAC [BZ13], full domain hash signatures [BR96, BDF+11], as well as the GPV08 identity-based encryption scheme [GPV08, Zha12].

For the moment, I have not merged this version into the main branch of EasyCrypt for mainly two reasons:

1. While I am very confident in the theoretical part of the paper, the implementation does not exactly match the theory. In fact, the implementation uses an encoding of the EasyCrypt language. This encoding has never been fully formalized, so the possibility of encountering a bug exists.

2. The language provided in the implementation is more restrictive than the one provided in the theory. This is too restrictive to express some fundamental lemmas like the O2H one [Unr15]. This is a severe restriction that needs to be removed.

To address these issues, we have started a collaboration with Dominique Unruh to develop a logic that is sufficiently expressive and closer to the implementation. This will be one of my main subjects in the coming years.

2.2 Main contributions

We developed the pRHL logic [68], which is a relational Hoare logic for verifying the equivalence of probabilistic programs [53, 64]. We implemented this logic in Coq, with the CertiCrypt library [68], which made it possible to verify numerous cryptographic constructions [73, 69, 62, 61, 54, 46]. One of the keys to success has been the intensive use of reflective evidence and the certification of program transformations. To facilitate the adoption of formal proofs by cryptographers, we have then developed the EasyCrypt proof assistant [60, 52, 55]. It is based on on a combination of proof assistants and SMT. EasyCrypt has been used for many applications [60, 51, 50, 49, 44, 31, 26, 19], with in particular the proof [19] of the SHA3 standard (partly funded by NIST) and the proof of the AWS KMS protocol [18].

Another aspect of my work is proof automation. We have implemented the first automatic synthesis tool for cryptographic primitive constructions [50, 47, 39, 24] and I helped develop variants of EasyCrypt with a more limited but automatic scope [39, 44].

The main theoretical contribution of this chapter is clearly the probabilistic Relational Hoare Logic. From the point of view of the tools the CertiCrypt library was a major step allowing us to formally define the semantic of probabilistic programs, to prove the rules of the logic, and to capture most of the notions needed to formalize cryptographic proofs. The second and most famous milestone is EasyCrypt, it has been used to develop mainly examples but it has also been used to develop a variant of the logic like apRHL. It is the key milestone of the verification tool chain that we use for the formal verification of Jasmin program in the Formosa project.

There are four major publications on that domain:

- In [68], we presented pRHL. It was also my first POPL paper.
- In [60] we presented the EasyCrypt tool, and the technique that can be used to simplify cryptographic proofs, the paper got the Best Paper Award.
- In [47], we presented ZooCrypt. This paper was an important step for my work on masking.
- In [12], we present a first extension of EasyCrypt allowing to perform proofs in the post-quantum setting.

Chapter 3

Cryptographic implementations

3.1 Problem with cryptographic implementation

Cryptographic software is pervasive in software systems. Although it represents a relatively small part of their code base, cryptographic software is often their most critical part, since it forms the backbone of their security mechanisms. Unfortunately, developing high-assurance cryptographic software is an extremely difficult task. Indeed, good cryptographic software must satisfy multiple properties, including efficiency, protection against side-channel attacks, and functional correctness, each of which is challenging to achieve:

- *Efficiency.* Cryptographic software must imply *minimal overhead* for system performance, both in terms of computational and bandwidth/storage costs. These are first-class efficiency requirements during development: a few clock cycles in a small cryptographic routine may have a huge impact when executed repeatedly per connection established by a modern service provider.

- *Functional correctness.* Specifications of cryptographic components are often expressed using advanced mathematical concepts, and being able to bridge the enormous semantic gap to an efficient implementation is a prerequisite for the implementor of a cryptographic component. Moreover, implementations may involve unconventional tasks, such as domain-specific error handling techniques. Guaranteeing functional correctness in these circumstances is harder than for other software domains, but it is critical that it is guaranteed from day one—contrary to the usual detect-and-patch approach—as implementation bugs in cryptographic components can lead to attacks[BBPV12, GK13].

- *Protection against side-channel attacks.* In-depth knowledge of real-world attack models, including side-channel attacks, is fundamental to ensure that the

implementation includes adequate mitigation. I will explain this in more detail in the next section.

Efficiency considerations rule out using high-level languages, since the code must be optimized to an extent that goes far beyond what is achievable by modern, highly optimizing compilers. Furthermore, there are concerns that highly optimizing compilers may introduce security flaws [DPS15, KPVV16]. As a consequence, the development of cryptographic software must be carried out at the assembly level and is entrusted to a few select programmers. Moreover, these programmers rely on rudimentary tooling that is often co-developed with the implementations themselves. For instance, security- and performance-critical parts of the OpenSSL library result from an *ad hoc* combination of pseudo-assembly programming and scripting, known as “perlasm”. Another alternative is to use the qhasm language [Ber], which simultaneously elides low-level details that are inessential for fine-grained performance tuning and retains all performance- and security-critical aspects of assembly programming. qhasm achieves an excellent balance between programmability and efficiency, as evidenced by a long series of speed-record-breaking cryptographic implementations. Due to their nature, these approaches do not lend themselves to being supported by formal verification.

Functional correctness and side-channel security requirements for high-assurance cryptography impose going significantly beyond the current practices used for validating implementations, namely code inspection, code testing (and in particular, fuzzing), and even static analysis. Code inspection is time-consuming and requires a high level of expertise. Testing is particularly effective for excluding errors that manifest themselves frequently but performs poorly at detecting bugs that occur with very low probability. Static analysis is useful for detecting programming errors but does not discover functionality bugs. A better alternative is to create machine-assisted verification frameworks that can be used for building rigorous proofs of functional correctness and side-channel security. However, these frameworks are not easily applicable to assembly languages, which is the level at which guarantees need to be provided.

In the realm of cryptographic primitives, especially with the emergence of new post-quantum primitives, proving functional correctness is not an end in itself. When the specification involves a relatively simple operation like exponentiation, it may be convincing that it accurately represents the intended mathematical concept. However, it is preferable to go beyond conviction and prove that the specified operation indeed satisfies the expected properties. The challenge arises when the specification becomes more intricate, as observed in cases like Kyber and Dilithium. How can one trust a complex specification? More importantly, how can we establish the correctness of the specification itself?

For cryptographic primitives, like encryption schemes or signature schemes, there

is a well-established setting to describe the properties that should be satisfied. So my claim is that we should prove functional correctness against a specification that has been itself proved to be cryptographic secure, as can be done using a tool like EasyCrypt.

3.2 Proposed solutions

We have pioneered the development of the Jasmin infrastructure, aimed at facilitating the creation of efficient, correct, and secure cryptographic code. Our primary objective is to deliver formally verified implementations capable of supplanting unverified cryptographic routines present in conventional libraries. The idea is to have a programming language designed to serve both the programmer—by providing expressiveness and control—and the tools for analyzing and transforming programs. The Jasmin programming language combines high-level functionality with instructions similar to those in assembly languages (in particular, vectorized instructions).

3.2.1 Efficiency

Rather than relying on a compiler that aggressively optimizes code generation for efficiency, our approach is to craft a language enabling programmers to write code that can be seamlessly compiled into efficient assembly programs. This approach offers two distinct advantages: firstly, it simplifies the compiler development process (including its formal verification), and secondly, it empowers programmers with greater control over the resulting compiled code. This latter aspect is particularly crucial in scenarios where efficiency isn't the sole objective, and considerations for security assurances are paramount.

To this end, the Jasmin language provides access to the low-level functionalities of assembly. In particular, the programmer has access to most assembly instructions and flags (such as the carry flag). The programmer exercises strong control over how the code is generated:

- They can specify whether a variable should be stored on the stack or in a register.

- They can determine which functions should be inlined, which loops should be fully unrolled, etc.

- Direct access to most assembly instructions is available, providing similar functionality to C intrinsics. Notably, X86-64 AVX and AVX2 instructions are directly usable from Jasmin source programs.

In this sense, Jasmin can be viewed as a macro language for assembly.

Jasmin also provides higher-level structures like arrays. Again, the programmer can decide whether the array should be stored on the stack or in registers (in which case, the compiler will allocate each cell of the array into registers). While compilation of arrays corresponds to in-place memory access, their source semantics are functional; i.e., updating an array t returns a fresh t' while the original array t remains unmodified. It is the role of the compiler to ensure that all memory operations can be performed in place. This complicates the compiler, but it offers significant advantages when the goal is to prove the functional correctness of Jasmin programs or to build static analyses for Jasmin: there is no alias.

3.2.2 The certified Jasmin compiler

The compiler is predictable and produces efficient code. However, ensuring the correctness of the resulting assembly code is paramount. How can we confidently assert that the compiler doesn't inadvertently introduce bugs during the translation process? Our answer is to prove the functional correctness of the compiler using the Coq proof assistant, like it has been done with the CompCert compiler [Ler06].

An overview of the Jasmin compiler is depicted in Figure 3.1. Nearly all components of the Jasmin compiler are directly written and verified in Coq (highlighted in green). The remaining components (highlighted in blue) are implemented in OCaml, with a Coq-verified checker ensuring the correctness of their transformations. Ultimately, the Coq code representing the compiler is extracted to OCaml, yielding an executable compiler.

An intriguing aspect to note is the minimal use of intermediate representations (Jasmin, Stack, Linear, ASM) in the compilation chain. Both Jasmin and Stack share identical code representations, with only the semantics differing. This streamlined approach is made possible by Jasmin 's capacity to address low-level features directly at the source level.

The unverified passes, depicted in brown, include the initial parsing pass followed by the *Preprocess* pass. The latter constructs the data structure (the Abstract Syntax Tree - AST) employed by the certified compiler. Additionally, it elaborates on the implicit information not present in the Jasmin syntax (.jazz) and potentially conducts sanity checks on the resultant Jasmin program.

The third pass entails type checking, which doesn't alter the program. Although this phase remains unverified currently, its certification holds significance, as elaborated in Section 3.2.3.

After completing these initial stages, one can proceed with compilation using the certified compiler or choose to conduct various verifications on the resulting Jasmin program. The first verification step is a safety analysis, aiming to ensure the program's well-defined semantics (refer to Section 3.2.3). The second step involves type

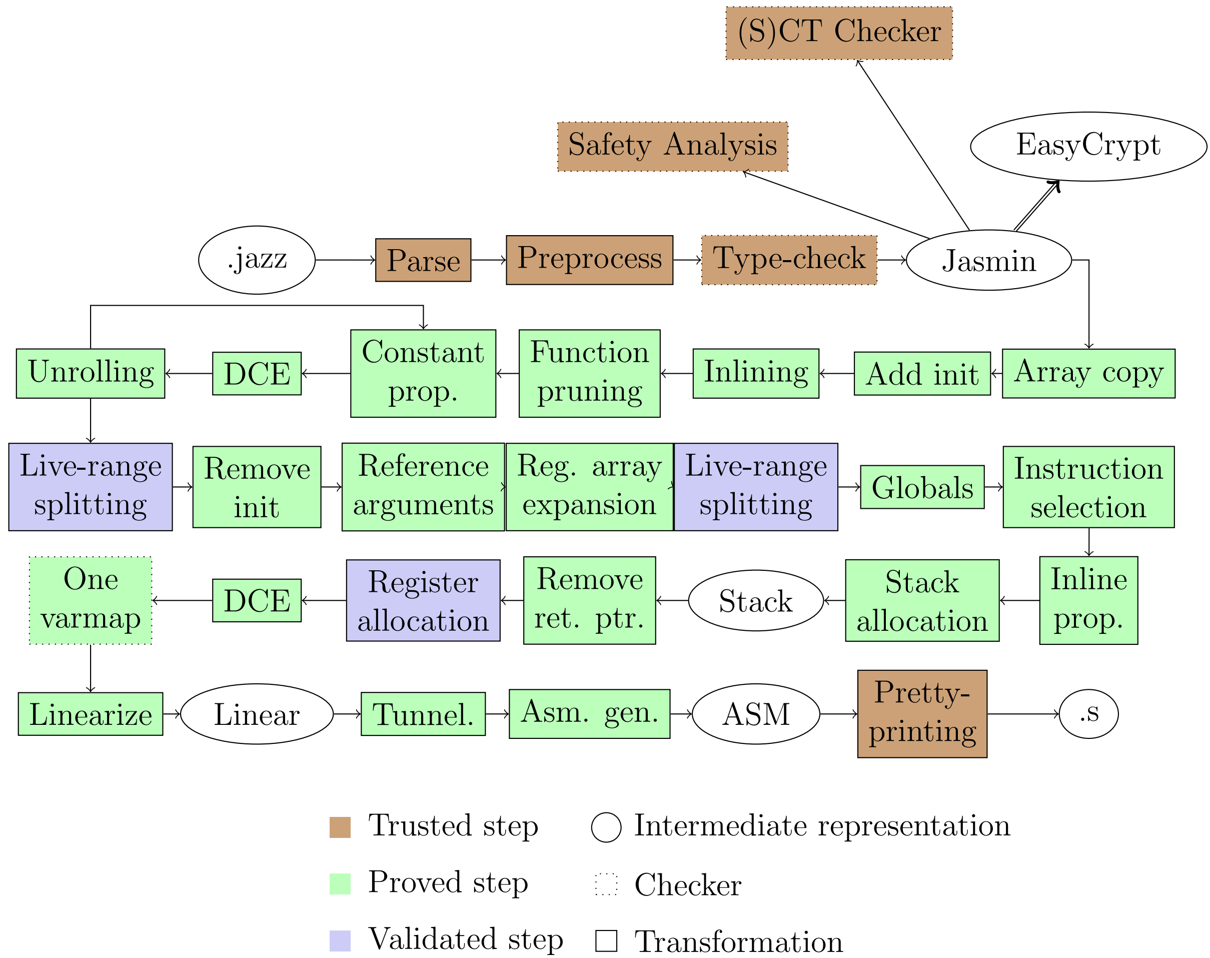


Figure 3.1: Overview of the Jasmin compiler

checkers to verify that the program is (speculatively) constant time (see Sections 4.2 and 4.3). The last step involves translating Jasmin programs into EasyCrypt, enabling verification of properties such as functional correctness (Section 3.2.4), constant-time behavior (Section 4.2), and security (Section 3.2.4). Importantly, these verifications and translations occur just before the fully verified compiler phases.

Finally, the last pass (Pretty-printing) is unverified. It prints assembly code and outputs it to a file. While proving its correctness is important, it poses challenges in Coq due to the inability of Coq functions to include side effects. Fortunately, manual inspection of this pass is relatively straightforward[1].

Let's take a look at the general form of the functional correctness of the compiler:

Theorem 1 (Correctness)**.** *For all source programs P, target programs P', source states s and s', and target state t, such that:*

1. *The compilation of P succeeds and generates P'.*
2. *The evaluation of P starting from s leads to the final state s': $s \xrightarrow{P} s'$.*
3. *The state s is in relation with the target state t: $s \approx t$.*

Then there exists a target state t' such that:

- *The evaluation of P' starting from the target state t leads to a state t': $t \overset{P'}{\rightsquigarrow} t'$.*
- *The state s' is in relation with the target state t': $s' \approx t'$.*

I will not describe the exact definition of $\approx$ here. What is important to remember is that this relation allows us to lift functional properties of Jasmin programs from source to assembly. This is because the relation ensures equality of the results between the source and target programs (i.e., return values and memory). Note that this theorem guarantees only the existence of a target derivation that satisfies the relation with the source. The generalization to all target executions follows from the safety of the source program and the determinism of the target language. Assume we have a target derivation $t \overset{P'}{\rightsquigarrow} t'$ By the safety of the source, there exists a source state s' such that $s \xrightarrow{P} s'$. According to Theorem 1, there exists t'' such that $t \overset{P'}{\rightsquigarrow} t''$ and $s' \approx t''$. Given the determinism of the target language, it follows that $t' = t''$.

Unfortunately, this theorem is not sufficient to prove the preservation of relational properties, such as cryptographic constant time (see Section 4.2 and Section 4.3).

[1] We can certainly improve the code for better readability.

3.2.3 Type checking and safety analysis

It is important to note that Theorem 1 guarantees correctness only for well-defined programs (condition 2), meaning those with clearly defined semantics. In strongly typed programming languages such as OCaml or Rust, safety is enforced by the type system, although certain primitive operations may still require runtime checks to prevent violations. For instance, array access necessitates dynamic verification to avoid buffer overflow, while division operations must ensure that the divisor is not zero.

In the context of Jasmin, we strive to avoid these runtime checks for efficiency purposes. However, the basic type system of Jasmin is insufficient for this task, necessitating a more complex static analysis known as *safety analysis*. Its goal is to ensure there are no occurrences of buffer overflow, division by zero, or the use of uninitialized variables (or accessing unallocated memory).

To address this need, Adrien Koutsos and Vincent Laporte have developed a safety analysis tool in OCaml specifically for Jasmin programs. This tool assumes that the input program is well-typed. The output of the analysis consists of safety preconditions that must be verified by the initial state of the program. These preconditions may specify requirements such as memory regions being readable/writable by the program, with regions defined in terms of the program's arguments (e.g., a pointer p and length n defining a memory region between p and $p + n$)

However, the current state of the safety checker presents several challenges:

- **Certification/Validation**: While Theorem 1 relies on safety, neither the safety checker nor the type system are certified. While proving the type system's correctness may be relatively straightforward, certifying the safety analysis is likely beyond the current capabilities of program verification technology. In particular, this will require a certified version of the Apron [JM09] static analysis library. However, validating the analysis results may be feasible using techniques similar to those used in [BJPT10, BCJP09, FMP13].

- **Consistency with Formal Semantics**: The conditions verified by the safety analysis are strongly tied to the formal semantics of Jasmin, yet its implementation only considers Jasmin's AST. Maintaining consistency between the analysis and semantics requires manual inspection, which is a tedious and error-prone process.

- **Genericity**: Although Jasmin supports back-ends for X86-64 and ARM-v7, the current safety analysis only works for X86-64. This limitation stems from the analysis's development predating the introduction of the ARM back-end. Achieving a more backend-independent analysis approach is challenging due to the direct usability of low-level assembly instructions at the Jasmin source level.

- **Completeness/Efficiency**: The tool's completeness is limited, and its execution can be prohibitively costly. Addressing this issue involves improving the tool itself or relaxing Jasmin's semantics to simplify the conditions required for verification. However, this reintroduces the challenge of maintaining consistency with the formal semantics.

In summary, while the safety analysis tool is a crucial component for ensuring program safety in Jasmin, several challenges remain in terms of certification, consistency, genericity, and efficiency. Now, let's delve into potential strategies to address these diverse hurdles.

Refining Jasmin's semantics From a safety perspective, a notable challenge with the current semantics lies in its reliance on big-step semantics. This approach brings forth two significant implications. Firstly, only terminating programs are considered safe, and so safety analysis must ensure program termination. Determining termination is notoriously difficult, and furthermore, the termination condition imposed is overly restrictive for Jasmin programs. For instance, cryptographic primitives like Kyber and Dilithium employ a rejection sampling algorithm, i.e., repetitive sampling until a specific condition is met to ensure adherence to a desired distribution[2]. Secondly, our big-step semantics fail to differentiate between non-terminating programs and those that raise runtime errors (such as type errors or buffer overflows, for example). In both scenarios, the program lacks a formal semantic representation. Consequently, expressing the correctness of the type system is not possible because we cannot express that no runtime error occurs. In fact, it is possible to address this for the semantics of the expressions of the language; their semantics is defined by a function that can return an error, but not for the instructions of the language.

Hence, the objective is to revise the Jasmin semantics to eliminate the termination constraint and accommodate runtime errors. The conventional approach involves transitioning to a small-step semantics, albeit at the cost of rewriting the compiler proofs entirely, which will be more complex since big-step semantics offer a straightforward induction principle. Certainly, leveraging interaction trees [XZH+19] presents a promising solution. This framework facilitates capturing non-terminating semantics while preserving an induction principle akin to our current methodology. This alignment is crucial for retaining the extensive 80k lines of proofs associated with the Jasmin compiler.

Partial certification of safety analysis To advance towards the formal verification and validation of safety analysis, a promising approach involves dividing

[2] For example, to uniformly sample a number between 1 and 4, one might roll a six-sided die until a result of 4 or lower is obtained, yielding a uniform distribution within the range [1..4].

the safety analysis into two distinct components. Initially, this entails developing a safety condition generator—a tool that computes, for each program point, a set of conditions that need to be satisfied to ensure overall program safety.

Assuming that the termination problem is solved, verifying the correctness of both the type checker and the safety condition generator becomes straightforward: *If the program is well-typed, then it undergoes evaluation without encountering type errors. Moreover, if it is well-typed and the generated safety conditions are satisfied at each program point, then instructions can be evaluated without triggering runtime errors.*

It would be advantageous not only to prove the correctness of the safety condition generator but also its completeness. This will ensure that the unsatisfiability of a safety condition leads to a runtime error. Moreover, this division introduces greater generality into safety analysis. Rather than needing to know every instruction to determine which conditions must be met, the analysis simply needs to interpret and validate conditions generated by the generator.

In the long term, a significant project involves validating the outcomes of safety analysis using techniques akin to those outlined in [BJPT10, BCJP09, FMP13].

Genericity The lack of generality in safety analysis primarily stems from two factors, both related to the direct access to target-dependent low-level instructions in Jasmin. Firstly, for all low-level instructions, the analysis must ascertain which conditions must be satisfied. This challenge can be addressed through the use of a generator, as detailed in the preceding paragraph. Secondly, the analysis must understand the semantics of instructions to monitor the values of certain variables[3], typically those involved in array access to ensure they remain within bounds.

In the context of Jasmin, the number of low-level instructions that are accessible is relatively large and, furthermore, regularly increases because users request the addition of some specific instructions required to implement their algorithms efficiently.

A potential solution to this issue involves defining a concise language to describe instruction semantics, comprising a small set of primitive operations. Consequently, the safety analysis would only need to understand and handle this compact language, eliminating the need for constant extension or patching whenever new instructions or backends are introduced. Note that defining such a language would be very useful in other parts of the Jasmin infrastructure. The semantics of the instructions are defined in Coq (for the formal definition of the semantics), and they are also defined in EasyCrypt (which is needed to prove properties on Jasmin programs), and partially in the safety analysis (for the previous reasons). This language would allow them to

[3] For most variables, it is not necessary to precisely keep track of their value because they are not involved in any safety conditions.

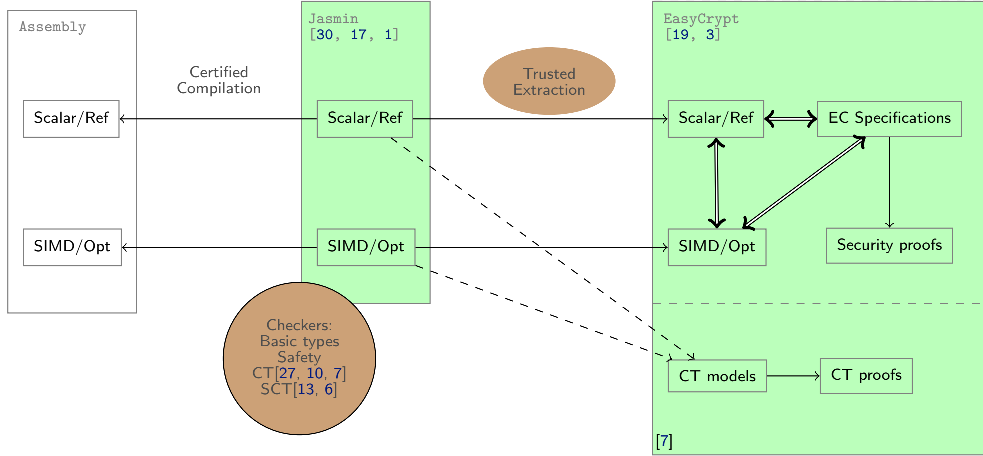


Figure 3.2: The Jasmin/EasyCrypt framework

be defined only once and then automatically define the Coq and EasyCrypt semantics from it.

3.2.4 Proving **Jasmin** programs

Our initial approach to verifying Jasmin programs involved the development and proof using Coq of a Hoare logic or weakest precondition calculus tailored to Jasmin programs. This method offered the significant advantage of maintaining a small trusted computing base, primarily consisting of Coq and our formalization of the assembly language. Integration of tools like Iris [JSS+15] would have further bolstered this approach.

However, we have opted for a different strategy: translating Jasmin programs into EasyCrypt programs. While this approach does introduce EasyCrypt and the translation process into our trusted computing base, it offers several compelling advantages. This enables us to verify properties of interest using various EasyCrypt logics. Additionally, this approach empowers us to verify the cryptographic security of our Jasmin implementations, further enhancing the reliability and integrity of our system.

We now have various methods for specifying and verifying our implementations. The first approach is to utilize traditional Hoare logic, a method adopted by many tools[4] such as Why3 [BFMP11], Dafny [Lei10], and F* [SHK+16]. To utilize this approach, we begin by crafting a functional or mathematical description of the

[4]Not all the tools are based on Hoare logic, but they all share the requirement of having a functional specification.

computation our program is intended to perform. This description allows us to express both the post-condition, defining what needs to be satisfied at the end of the program, and the precondition, specifying what must be satisfied before the program's evaluation. Subsequently, we can employ Hoare logic or any equivalent method to demonstrate that our program meets its specification.

The second method involves employing program equivalence. In this scenario, the specification itself takes the form of an imperative program, as commonly seen in Request for Comments (RFC). Functional correctness is then equated with program equivalence between the implementation and its specification. Here, equivalence implies that if inputs are related, the outputs will be as well. Fortunately, EasyCrypt shines in proving program equivalence, leveraging the robust pRHL logic.

These approaches lead to intriguing possibilities. Firstly, since the specification is now an EasyCrypt program, we can verify its semantic security. This is particularly crucial for complex algorithms like Kyber and Dilithium, where manually verifying that the formal specification aligns with the RFC can be as challenging as verifying the code of a reference implementation. Moreover, how can we be sure that the RFC makes sense? By proving the semantic security of our specification, we do not establish that it corresponds to the RFC but a stronger result: our specification is secure. Then, by proving program equivalence between the specification and implementation, we ensure the semantic security of the implementation. Thus, even if the implementation deviates from the RFC, we have assurance of its semantic security. Furthermore, passing test vectors provides informal assurance of its interoperability with other implementations.

Secondly, leveraging program equivalence offers a novel approach to validating optimized implementations. Often, optimization entails utilizing a vectorized version of the program, where independent computations are executed in parallel. This is facilitated by SIMD instructions, enabling multiple data values to be processed with a single instruction. A prime illustration of this concept is ChaCha20, a block cipher employed in counter mode.

In the encryption process of ChaCha20, each message block is encrypted using a secret key, a nonce, and a counter (unique for each block) passed to the block cipher. The encrypted block results from XORing the message block with the output of the block cipher. Notably, each iteration of the loop in this process operates independently, allowing for potential parallelization.

Validating the correctness of an optimized implementation can be achieved through incremental steps. Initially, the program is transformed to execute eight iterations of the loop body at each iteration of the main loop; since this transformation can be justified in a generic way (independent of the body). This restructuring results in two nested loops. While the control flow of the program alters, the underlying data structure and computation order remain consistent.

Let us now focus on the internal eight iterations of the loop body. Subsequently, this inner loop is unrolled, and since each iteration is independent, intermediate variables used for computation can be renamed, yielding eight variables for each original variable. Proving program equivalence at this stage is straightforward. With each loop iteration operating on disjoint variables, it becomes feasible to reorder instructions to interleave the eight iterations of the loop body. Finally, it can be demonstrated that performing eight consecutive operations on independent variables is equivalent to executing the corresponding single SIMD instruction. Through transitive reasoning, equivalence between the reference implementation and its optimized vectorized counterpart is established.

While this technique proves effective in many cases, there are instances where it is impractical to employ. Particularly, when the algorithm for exploiting SIMD instructions diverges significantly from the reference implementation, it may be more straightforward to directly verify functional correctness using techniques like Hoare logic.

Figure 3.2 illustrates the structure of the Jasmin/EasyCrypt framework. At its core lies the Jasmin language, which accommodates both reference/scalar implementations and optimized/SIMD implementations for each primitive. These implementations undergo scrutiny through the type system and the safety checker. Additionally, there's the option to utilize either the constant-time type checker (detailed in Section 4.2) or the speculative constant-time checker (expounded in Section 4.3) to fortify resilience against (speculative) side-channel attacks.

Thus, implementations are translated into assembly and extracted into EasyCrypt for verification of both functional correctness and semantic security. The safety, along with the compiler's correctness, guarantees that both the source code and its assembly counterpart produce identical results. Furthermore, safety ensures that the source code and the extracted EasyCrypt program share congruent semantics. On the EasyCrypt side, a specification is crafted specifically tailored for semantic security proofs. This often involves abstracting certain implementation details; for example, in verifying the semantic security of a scheme like Elgamal, the focus is on ensuring that the underlying structure adheres to properties expected by cyclic groups, while the specific implementation details of the cyclic group are deemed irrelevant.

Subsequently, the functional correctness of the reference implementation is proven against the EasyCrypt specifications. The functional correctness of the optimized version is proved either through establishing program equivalence with the reference implementation or directly against the EasyCrypt specification.

This comprehensive chain of validation allows for the assertion that the assembly codes are not only functionally correct but also semantically secure.

3.3 Main contribution

A important contribution of this work is the design of the language itself. It allows to reconciliate highly optimized programs at assembly level and formal proofs. The associated verification tools ensure that Jasmin programs are safe, functionally correct and resistant to attacks by auxiliary channels (see Sections 4.2 and 4.3). The formal proof of the Jasmin compiler ensures that those properties are preserved at the assembly level.

The verification tool chain is based on EasyCrypt, which makes it possible to prove semantics security of the algorithms.

The first implementations [30, 19, 17] of the ChaCha20, Poly1305, curve25519, SHA3 primitives show that the code issued by the Jasmin compiler is as efficient as the assembly code written by hand in the OpenSSL library. The most emblematic implementations are that of SHA3 and Kyber: implementations for which the four properties have been established [19, 3]. Coq certification of the compiler ensures that functional correctness is preserved.

The final result of this work is the libjade developed in the Formossa project. libjade is a formally verified cryptographic library written in the Jasmin programming language with computer-verified proofs in EasyCrypt. The primary focus is on offering high-assurance implementations of post-quantum cryptographic primitives to support the migration to the next generation of asymmetric cryptography.

Chapter 4

Side Channels

4.1 Problem with side channel attacks

Being able to establish proofs of security at the algorithmic level of cryptographic primitives, and to link these algorithms with their implementations, is already a major step forward and provides guarantees that are unrivaled. However, this is still not enough. The main problem lies in the discrepancy between the adversary model used in the security proof and the real power or capability of adversaries attempting to attack the code in practice.

Security proofs are generally conducted in a black-box model, where the adversary can see or control some of the inputs provided to the primitive and observe outputs (essentially everything that passes through the network). However, the adversary has no information about how the primitive is evaluated.

While there are scenarios where this black-box model is reasonable, in most cases, the adversary can gather significant information through hidden channels. The amount of information they can access often depends on their ability to interact with the machine executing the code.

For example, if the adversary is able to run code on the same computer as the one executing the cryptographic primitive, it is crucial to ensure that the observable timing behavior of the compiled program does not leak sensitive information. Failing to address these concerns creates a significant attack vector against cryptographic implementations [Ber05, AP13].

Indeed, one prevailing view is that critical code must adhere to the *cryptographic constant-time* (CCT or simply CT) discipline, meaning its control flow and sequence of memory accesses should not depend on secrets [Ber05]. High-assurance cryptographic software must be guaranteed to follow this discipline correctly. Unfortunately, compiler optimizations can break CCT.

While necessary, this countermeasure is not sufficient. At the abstraction level

provided by hardware architecture, programs are assumed to execute sequentially in the order dictated by the control flow. However, at the hardware implementation level, program execution is much more complex, involving out-of-order and speculative execution. This complexity at the microarchitectural level was meant to be transparent to developers, who should only need to reason about programs using the abstractions provided by the architecture.

Yet, Spectre attacks (soon followed by many others, e.g., [BMW+19, BMS+20, SSL+19, RMR+21, RBBG21]) revealed how attackers could exploit speculative execution to exfiltrate secrets that were otherwise well-protected at the architectural level. The consequences of such speculative attacks can be devastating. While Meltdown is often viewed as a hardware bug and can be fixed in future generations of processors, Spectre is not a hardware bug—it's a feature. Therefore, we must develop software countermeasures to defend against it.

When the adversary has physical access to the machine, he can measure power consumption [KJJ99], electromagnetic radiation [GMO01], or even the noise produced by the computer's fan [GSE20]. These sources of information can be exploited to mount side-channel attacks, such as Differential Power Analysis (DPA). Protecting against such attacks is more challenging, as any intermediate result can potentially leak sensitive information, not just conditional branching or memory accesses.

The most widely deployed countermeasure against these physical side-channel attacks is called *masking* or *secret sharing*. The concept involves splitting a secret into several shares, often using random data, such that the sum of the shares corresponds to the original secret. This forces the adversary to gather all the shares to recover the secret value.

In this chapter, I will provide an overview of the steps we have taken to ensure that cryptographic implementations are resistant to these various types of attacks.

4.2 Preservation of Constant Time

One approach to protecting cryptographic libraries from cache attacks is by adhering to the constant-time (CT) discipline. In theory, this discipline is straightforward: a program is considered constant-time if its control flow and memory accesses do not depend on secret values. However, writing efficient constant-time code is notoriously error-prone.

Moreover, most tools designed to verify compliance with the CT discipline operate at the source code level, whereas this property ultimately needs to be guaranteed at the assembly level. This raises a critical question: Does the compiler preserve the constant-time property during the translation from source code to assembly?

The answer is not simple. While one might assume that compilers are extensively

tested for correctness, such testing does not necessarily ensure that the generated code adheres to the constant-time discipline. Furthermore, generating constant-time code is generally beyond the scope of general-purpose compilers like clang and gcc.

When we aim to formally establish that a compiler preserves the CT discipline, a second question arises: How can the CT discipline be formally defined?

A common methodology involves using instrumented semantics, where each step of execution is modeled to potentially generate some form of leakage. For instance, the rule governing conditional instructions[1] might be represented as follows:

$$\{\mathsf{if}\ e\ \ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \rho\} \xrightarrow{b} \{c_b, \rho\}$$

where $b = [\![e]\!]_\rho$ corresponds to the evaluation of the conditional expression e in the state ρ. Similarly, the evaluation of load and store instructions leaks the accessed address, while other instructions may leak nothing[2].

The notion of CT is generally parameterized by a relation ϕ between initial states, which usually describes which parts of the state contain public data (in that case, the relation ensures equality in both states) and which contain secret data. We say that a program p is ϕ-CT if, for any pair of initial states (ρ_1, ρ_2) in relation ϕ, and for any number of execution steps n:

$$\left.\begin{array}{l} \{p, \rho_1\} \xrightarrow{L_1}{}^{n} \{c_1, \rho\} \\ \{p, \rho_2\} \xrightarrow{L_2}{}^{n} \{c_2, \rho\} \end{array}\right\} \Rightarrow L_1 = L_2$$

In other words, the generated leakages after n steps of execution are equal[3].

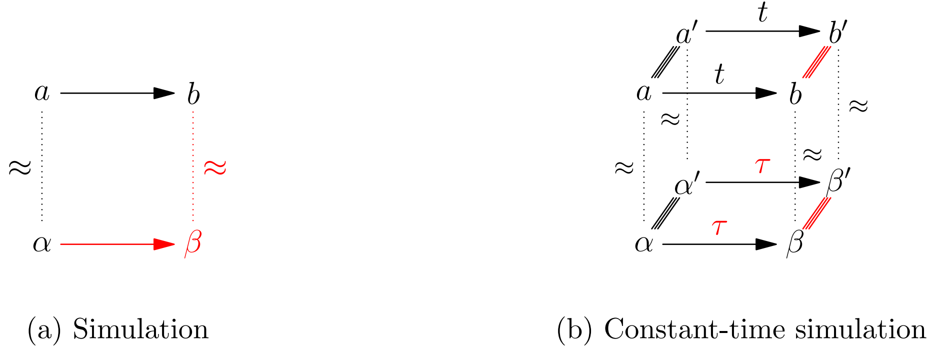
Proving the preservation of CT requires establishing that if the source program is CT with respect to ϕ, then its compiled version is also CT. Figure 4.1 illustrates the core idea of the method introduced in [27] to establish that a compilation step preserves constant time. Figure 4.1a recalls the standard notion of simulation from compiler verification. In the simplest (lockstep) setting, one requires that the simulation relation relates one step of execution of a source program S with one step of execution of its compiled version C, where black represents the hypotheses and red represents the conclusions. The horizontal arrows represent one step of execution of S from state a to state b, and one step of execution of C from state α to state β. The relation $\cdot \approx \cdot$ relates the execution states of the source and target programs.

While traditional simulations are established through 2-dimensional diagram chasing, constant-time simulations require 3-dimensional diagram chasing. Figure 4.1b illustrates the concept of constant-time simulation in the lockstep case,

[1] A while loop is a particular case of a conditional instruction.

[2] Some models also take into account timing-dependent instructions like division.

[3] Remark: Defined this way, this notion makes sense only for safe programs, i.e., programs that cannot get stuck before the end of execution. For simplicity, I will only consider safe programs here.



(a) Simulation

(b) Constant-time simulation

Figure 4.1: Lockstep simulations

introducing the relations $\cdot \equiv_S \cdot$ and $\cdot \equiv_C \cdot$ between source and target states, depicted by triple lines in the diagram. Horizontal arrows represent single-step executions as before, but now we consider two executions at the source level and two at the target level.

To construct this 3-dimensional diagram, one can assume all relations depicted in black (e.g., equality of the source leakages) and must establish all relations depicted in red (e.g., equality of the target leakages, and preservation of the relations ($\cdot \equiv_S \cdot$ and $\cdot \equiv_C \cdot$). The relations $\equiv_S$ and $\equiv_C$ serve a similar role to ϕ for the initial state but must be invariant throughout the execution.

Extending this diagram to multiple steps of execution is straightforward. The major advantage of this technique is that it does not require modifications to the existing functional correctness proof (the simulation) of the compiler. However, proving constant-time simulations can be tedious due to the 3-dimensional nature, where four executions must be related.

Moreover, many compilation steps cannot be proven using a lockstep simulation diagram, where one source step corresponds directly to one target step. Fortunately, our methodology can be extended to handle multiple steps of execution.

While the proofs presented in [27] were formally verified using Coq, they were conducted for a toy compiler. In [15], we extended this work and demonstrated that the methodology can scale to a realistic C compiler: CompCert. Furthermore, in [10] and in the thesis of Swarn Priya, we applied a different methodology to prove that the Jasmin compiler preserves constant-time (CT) properties.

The key innovation in this approach is the removal of the overhead associated with the 3-dimensional diagram[4] by demonstrating that the target leakage is a function of the source leakage.

Figure 4.2 illustrates the core concept of the technique: we must demonstrate the

[4]Initially, we were quite excited by this elegant cube structure, reminiscent of Henk Barendregt's λ cube, but we soon realized it was not as convenient to use as we had hoped.

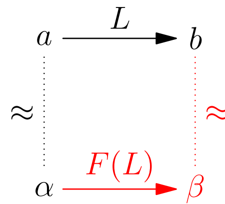


Figure 4.2: Lockstep simulations with leak transformer F

existence of a leak transformer F that allows us to establish the usual simulation diagram, this time accounting for leakage. It's important to note that the leak transformer depends solely on the leakage and not on the initial source and target states $(a,\ \alpha)$. As a result, proving the preservation of constant-time properties becomes straightforward since the target leakage is determined only by the source leakage[5].

To apply this technique to a realistic compiler, the leak transformer may sometimes need to depend on more information than just the source leakage. For instance, in the case of Jasmin, the leak transformer also depends on the value of the stack pointer. Fortunately, the stack pointer's value is determined solely by its initial value and the control flow followed by the program, which is uniquely defined by the prior leakage. Therefore, the leak transformer must take into account the initial value of the stack pointer. This requirement forces us to modify the initial state relation ϕ to ensure that the stack pointer's value is consistent in both states.

The main challenge in defining the leak transformer F is to accurately map each part of the leakage to the corresponding part of the evaluation. This is particularly challenging when the leakage is represented as a flat list. We addressed this issue by adopting a structured representation of the leakage that mirrors the tree structure of the evaluation.

4.2.1 Establishing Constant time

Having established that the compiler preserves constant-time properties, the next question is how to verify that a source program itself adheres to constant-time principles. We have developed two distinct methods to address this.

The first method is a generalization of the Volpano and Smith type system for non-interference [VIS96]. This type system, implemented in OCaml, supports type inference and allows us to determine whether a program respects constant-time constraints. As usual, it is not complete but actually all the code of libjade is

[5]Ensuring the safety of the source program is necessary, but this is standard practice when using forward simulation diagrams.

typable[6].

The second method leverages the fact that the constant-time property is a relational property between a program and itself. Since EasyCrypt is particularly well-suited for proving relational properties between programs, it forms the basis of this approach. We have defined an instrumented version of the extraction from Jasmin to EasyCrypt, which accumulates generated leakage in a data structure. This allows users to employ pRHL to prove that the leakage is independent of any secrets.

The primary advantage of this second technique is its flexibility compared to using a type system. For example, the leakage model can be easily adjusted to assume that memory accesses leak only the cache line rather than the full address, or that certain operators, like division, leak only the logarithm of their arguments. This flexibility was particularly useful in our work presented in [7].

Another key advantage is that this method allows for the concept of probabilistic constant-time, where we can prove that the resulting distribution of leakage is independent of the secret.

4.3 Speculative Constant Time

As explained earlier, while programs are assumed to execute sequentially, the hardware implementation often involves complex mechanisms such as out-of-order execution and speculative execution. These complexities can give rise to Spectre-like attacks, which are constant-time attacks occurring in contexts where the original code was not designed to be executed.

There are several variants of Spectre vulnerabilities:

- **Spectre V1**, or Spectre-PHT (*Pattern History Table*): This is the original Spectre attack, triggered by the branch predictor, which allows the speculative execution of a branch of a conditional instruction (i.e., a conditional direct jump).

- **Spectre V2**, or Spectre-BTB (*Branch Target Buffer*): This variant involves speculation induced by the prediction of indirect jumps. Spectre-RSB (Return Stack Buffer) is a related variant based on speculation over indirect jumps caused by return instructions.

- **Spectre V4**, or Spectre-STL (*Store-to-Load*) and Spectre-SSB (*Speculative Store Bypass*): These variants involve speculation on dependencies between stores and loads, where certain loads can speculatively ignore preceding stores.

[6]Some declassifies are needed due to rejection sampling algorithm.

While these mechanisms allow the processor to backtrack once it realizes that it made an incorrect speculative choice, some microarchitectural states (such as the cache) are not restored for efficiency reasons. This oversight enables cache attacks based on speculative execution. Notably, Spectre-BTB and Spectre-RSB are particularly powerful, as they potentially allow the speculative execution of any code crafted by an attacker.

The initial approach to protecting against Spectre attacks was to block speculative execution. Although processors do not offer a direct flag to disable speculative execution, most provide a `FENCE` instruction that halts the speculative execution of subsequent instructions until all previous instructions have been fully fetched, executed, and retired. While this method can protect against Spectre V1 and V4, it is ineffective against Spectre V2, which requires a different mitigation strategy, such as retpolines. However, this original approach comes with significant performance costs.

It is crucial to recognize that speculative execution is a vital feature of modern processors and will not be eliminated in the future. Therefore, understanding how to write code that is protected against Spectre attacks without severely impacting performance is essential.

In the context of Jasmin, Spectre-BTB does not need to be considered, as the language does not allow indirect jumps. However, Spectre-RSB remains a possibility. In the paper [13], we introduced a type system designed to protect against Spectre V1 and V4. This protection relies on the use of the `FENCE` instruction to block speculative execution, but the type system is designed to apply it only when necessary—specifically, to prevent speculative leakage of secret data. This approach yielded relatively good performance results for the cryptographic code in libjade. However, it has two significant drawbacks.

First, accurately benchmarking the cost of using `FENCE` is challenging. Benchmarks are typically conducted in a highly controlled and stable environment, but in a scenario where multiple processes are running in parallel, the cost of a `FENCE` instruction could be significantly higher. Second, the correctness of the type system depends on the assumption that the program is speculatively safe, meaning that array accesses must be within bounds even during speculative execution. This introduces two additional costs. For users, programs must be instrumented to ensure speculative safety. For Jasmin developers, a mechanism is needed to verify that programs are indeed speculatively safe.

In the paper [6], we adopted a similar approach to protecting against speculative leakage of secret data but employed a different protection mechanism instead of `FENCE`. This mechanism was initially introduced in llvm, and is known as Speculative Load Hardening (SLH) [Car]. The fundamental idea behind SLH is to mask any data that could potentially leak during misspeculative execution. Our approach was

	Unprotected	Protected	Low level
1		mask = init_msf();	`FENCE`; mask = -1; //i.e all bits set to 1
2	if e then	if e then	
3		mask = upd_msf(e);	mask = CMOV(!e, 0, mask);
4		x = protect(x, mask);	x = x & mask;
5	*leak*(x)	*leak*(x)	
6	else	else	
7		mask = upd_msf(!e);	mask = CMOV(e, 0, mask);
	...	...	

Figure 4.3: Core idea of selSLH

to apply SLH selectively—only in situations where speculative leakage might depend on secret data. We refer to this approach as selSLH.

But how do we obtain and maintain such a mask? Initially, we can use a `FENCE` instruction to ensure that the processor is in a normal state at that point and set the mask to -1 (i.e., it will mask nothing during normal execution). To detect if the processor is about to execute a mispredicted branch, we can leverage the conditional move instruction, which is not subject to speculation.

To provide an abstraction layer for developers and to simplify the definition of the type system, we introduced three new primitives in Jasmin. Figure 4.3 provides an overview of how these primitives are implemented and used.

The first primitive is *init_msf* (initialize the misprediction flag), which is implemented by using a `FENCE` instruction followed by setting the mask to -1. The second primitive, *upd_msf* (update the misprediction flag), is used to track the mask during the execution of conditional instructions. For example, if the processor mispredicts the value of the test e on line 2 and starts executing the true branch, the instruction on line 3 will set the mask to 0. The third primitive, *protect*, is used to mask data (as seen on line 4). In this case, the leaked value on line 5 will be 0, ensuring it does not depend on any secret. If the processor correctly predicts the true branch, then the instruction on line 3 (resp. line 4) will not alter the mask (resp. the value of x), and execution will proceed normally.

This protection mechanism is highly efficient because it avoids the need for systematically blocking execution. Instead, it only delays the execution of protected data until the test e has been fully evaluated. By doing so, SLH mitigates Spectre V1 by converting control flow dependencies into data dependencies.

Notably, only data that may leak during speculative execution needs protection, and even then, only if this data could depend on a secret when a branch is mispredicted. This leads to the key idea of applying SLH selectively, targeting protection

where it is truly needed.

The type system must ensure that the mask used as the argument of *protect* accurately reflects the status of any misprediction. Additionally, it must track values that depend on secrets during both normal and speculative execution. Crucially, the system also needs to monitor out-of-bound array accesses (read/write) that may occur during mispredicted execution, as these could lead to loading a secret (while a public value is expected) or storing secret data in positions that should only contain public data.

To address these concerns, the type system maintains two distinct security types for each variable, one for normal execution and another for speculative execution. This approach generalizes the Volpano and Smith type system, enabling a more comprehensive tracking of speculative and non-speculative security concerns.

The implementation of the type system relies on polymorphic type variables and constraints, similar to the approach used by François Pottier [Pot01]. This design enables type inference and allows the required fixpoint for the static analysis to be determined in a single pass.

One might argue that studying the security of countermeasures against speculative attacks at the source level is questionable. One reason for this is that source languages typically don't provide developers with access to the low-level details of the generated code. For instance, in C, a variable might be stored in the stack (i.e., in memory) or simply in a register, but this detail is crucial for writing speculative constant-time code. This is because speculative execution of operations on registers is fully rolled back at the microarchitectural level, whereas memory operations are not. Fortunately, Jasmin allows developers to specify such details and guarantees that they will be respected in the generated code. Nevertheless, the issue of preserving speculative constant-time properties through compilation is both important and challenging.

Another important point to discuss is the speculation mechanisms addressed by Jasmin. As previously mentioned, Spectre V1 is already efficiently handled. Unfortunately, there is currently no software extension of SLH that protects against Spectre V4. However, many modern processors now provide a flag, such as SSBD for x86, that allows V4 to be disabled. While disabling V4 does incur a performance cost, our testing on libjade code suggests that this cost is acceptable. Spectre V2 can be divided into two different mechanisms. Jasmin is not vulnerable to the first one (Spectre-BTB) because the compiler does not generate any indirect jumps. However, it is potentially susceptible to Spectre-RSB since return instructions are used. To address this, we are working on a solution that replaces return instructions with a jump table. The jump table is implemented using conditional jumps, which we know how to protect using SLH, and direct jumps.

In conclusion, Figure 4.1 presents the benchmarks. These results demonstrate

Table 4.1: libjade benchmarks on Intel Core i7 11700K (most optimized implementation of each primitive). "plain": cycles without any Spectre protections; "+SSBD": with SSBD CPU flag set; "+SSBD+v1": with SSBD CPU flag set and v1 countermeasures from [6]; "+SSBD+v1+RSB": with full Spectre protection as described in this paper; "increase": relative increase in CPU cycles between unprotected ("plain") and fully protected (+SSBD+v1+RSB).

Primitive	Impl.	Op.	plain	+SSBD	+SSBD+v1	+SSBD+v1+RSB	increase %
ChaCha20	ref	128 B	768	794	822	820	6.77
	ref	1 KiB	5932	6098	6140	6130	3.34
	ref	16 KiB	94420	96926	97220	97228	2.97
	avx2	128 B	344	344	398	398	15.70
	avx2	1 KiB	1198	1202	1244	1246	4.01
	avx2	16 KiB	19040	19052	19066	19068	0.15
Poly1305	ref	128 B	138	142	180	178	28.99
	ref	1 KiB	1126	1130	1154	1154	2.49
	ref	16 KiB	17542	17548	17568	17570	0.16
	avx2	128 B	138	142	182	180	30.43
	avx2	1 KiB	670	672	720	718	7.16
	avx2	16 KiB	8942	8948	8990	8986	0.49
XSalsa20Poly1305	ref	128 B	1626	1648	1680	1678	3.20
	ref	1 KiB	7860	7916	7926	7926	0.84
	ref	16 KiB	113852	114990	114892	114880	0.90
	avx2	128 B	1206	1212	1250	1246	3.32
	avx2	1 KiB	3140	3142	3190	3188	1.53
	avx2	16 KiB	32598	32574	32604	32602	0.01
SHAKE256	ref	128 B ← 128 B	1176	1226	1242	1230	4.59
	ref	256 B ← 128 B	2274	2368	2386	2370	4.22
	ref	512 B ← 128 B	4454	4654	4670	4746	6.56
	ref	1 KiB ← 128 B	8824	9214	9238	9284	5.21
	avx2	128 B ← 128 B	1206	1324	1390	1390	15.26
	avx2	256 B ← 128 B	2334	2450	2534	2546	9.08
	avx2	512 B ← 128 B	4588	4700	4796	4826	5.19
	avx2	1 KiB ← 128 B	9102	9216	9400	9384	3.10
X25519	ref	smult	121300	125798	126252	126286	4.11
	mulx	smult	102848	104150	104424	104428	1.54
Kyber512	avx2	keypair	27676	28106	28040	28090	1.50
	avx2	enc	37050	38332	38876	38792	4.70
	avx2	dec	29302	30444	30590	30714	4.82
Kyber768	avx2	keypair	43432	45708	45860	46548	7.17
	avx2	enc	57006	59316	60028	60674	6.43
	avx2	dec	46138	48418	48532	49294	6.84

that the performance overhead is relatively low, particularly for AVX2 implementations. This is because AVX2 implementations store most private data in large registers (xmm), leaving scalar registers free and avoiding the need to spill them to the stack. As a result, public data (such as pointers and loop counters) can remain in registers and don't need to be protected.

Regarding the cost of each countermeasure, the largest overhead arises from the need to use SSBD. It's also worth noting that some examples show a large overhead, which occurs in cases requiring only a small number of cycles to execute. In these cases, most of the overhead comes from the initial `FENCE` instruction, which is required to initialize the mask.

It's important to emphasize that, unlike the work on preserving constant-time properties, most of this work has not been formally verified in Coq.

4.4 Masking

The models for CT (Constant-Time) and SCT (Speculative Constant-Time) assume a scenario where the attacker can run a process on the target machine. However, an even stronger model exists where the attacker has direct physical access to the target hardware. In this scenario, the attacker can launch attacks by measuring power consumption or using electromagnetic probes. These types of attacks are commonly referred to as Differential Power Analysis (DPA).

Various models exist to characterize this type of attacker, ranging from more realistic to those better suited for formal verification. Our work primarily focuses on the t-probing model, which, while not the most realistic, is the most suitable for formal verification. In this model, the attacker can place up to t probes during the program's execution, with each probe capturing the value of an intermediate computation.

A common countermeasure against this type of attacker is to split secret data into $t + 1$ shares. Since the adversary can only recover up to t of these shares, they will be unable to reconstruct the secret. To split a secret s into $t + 1$ shares, we sample t independent random values, $r_0, \ldots, r_{t-1}$, and set the final share as $r_t = s + \sum_{i \in [0,t)} r_i$, where the + operator corresponds to the XOR operation[7]. This results in $s = \sum_{i \in [0,t]} r_i$, meaning that $r_0, \ldots, r_{t-1}$ together form a sharing of s. Importantly, the distribution of any t-tuple of r_i is independent of s.

Each basic operation involving secret data must be transformed into a corresponding "gadget" that operates on the shares. For linear operations, such as addition (or XOR), this transformation is straightforward. Suppose $a_0, \ldots, a_t$ is a sharing of a, and $b_0, \ldots, b_t$ is a sharing of b. Define $c_i = a_i + b_i$ for each i. Then

[7]This technique can be generalized to any cyclic ring by setting $r_t = s - \sum_{i \in [0,t)} r_i$.

$c_0, \ldots, c_t$ forms a sharing of $a + b$, since $a + b = \sum_{i \in [0,t]} c_i$.

Implementing multiplication (AND) is more complex. The basic approach involves first computing the matrix of cross products, which results in $(t + 1)^2$ intermediate values, and then summing them to construct $t + 1$ shares of the result. For this last step, one can define $c_i = a_i * b_0 + \ldots + a_i * b_t$. This gives:

$$c_i = a_i * \sum_{i \in [0,t]} b_i = a_i * b$$

However, directly observing c_i and a_i could reveal information about b. Furthermore, the distribution followed by c_i when executing this gadget on fresh sharings of a and b is not independent of b. To address this issue, the idea is to use fresh intermediate randomness.

For simplicity, let's look of multiplication gadget introduced in [ISW03] for the case of two shares (i.e, $t = 1$):

$$\begin{aligned} c_0 &= (a_0 b_0 + r) + a_0 b_1 \\ c_1 &= (a_1 b_1 + r) + a_1 b_1 \end{aligned}$$

Using this, the distribution of any single observation depends at most of one share of each input.

Proving the functional correctness of these algorithms is relatively straightforward, as it mainly involves verifying Boolean ring equalities. However, the real challenge lies in proving their security.

The initial security notion was defined as follows: First, an initial sharing of the secrets is performed, during which the adversary cannot observe any values. Afterward, the masked implementation is executed, and all intermediate values are recorded. Finally, the attacker can select up to t of these intermediate values, but their view must remain independent of the original secrets.

A major drawback of this security notion is its lack of composability. Even if two programs are individually secure under this model, their composition may not be secure. Additionally, ensuring that any selection of t intermediate values remains independent of the secrets leads to an exponential increase in the number of t-tuples that need to be considered as t and the program size grow. This complexity is further compounded by the quadratic nature of multiplication gadgets. Therefore, having a security notion that supports composability is crucial.

In [40] and later in [21], we were the first to develop a formal method for the automatic formal verification of masked implementations, leading to the creation of the tool [maskVerif]. The first key idea is to observe that successive applications of the optimistic sampling rule, as presented in Section 2.1, are sufficient to prove that a t-tuple is independent of certain secret values (by repeatedly applying the rule

until the secret values no longer appear). Although this approach is incomplete, it works surprisingly well in practice.

The second key insight is that if a k-tuple is independent of secret values, then any of its sub-tuples is also independent. This observation leads to an efficient divide-and-conquer algorithm. Suppose we want to prove that any t-tuple of expressions drawn from a set E is independent of some secret values. First, we try to find a subset A of E, as large as possible, such that its corresponding tuple is independent of the secret. This ensures that all t-tuples from A are independent of the secret. Next, we need to prove that all tuples from $E \setminus A$ are also independent, as well as any mixed tuples containing elements from both A and $E \setminus A$. This can be done by selecting one element from A and $t-1$ from $E \setminus A$, then two from A and $t-2$ from $E \setminus A$, and so on.

To extend this into a practical algorithm, a generalization is required. The algorithm takes as input a list of pairs, where the first element of each pair is a set, and the second element is a number indicating the minimum number of elements that should be taken from the set to build a tuple. For example, $(A_1, n_1); (A_2, n_2)$ indicates that we want to check whether all t-tuples composed of n_1 elements from A_1 and n_2 elements from A_2 are independent of the secret, where $t = n_1 + n_2$.

Although the complexity of the algorithm remains high, this approach allows verification of multiplication gadgets up to order 10, and even a full AES S-box for smaller orders. Unfortunately, this method does not scale to full implementations, where a compositional approach is required.

As previously mentioned, the primary challenge with composition arises from the definition of t-probing security itself- specifically, the requirement for an initial perfect sharing of secrets. This condition can be relaxed by requiring that any t-tuple of observations can be simulated using only t shares of each input, regardless of how the input sharing is performed. We have termed this new security notion *non-interference* (t-NI).

It is important to note that t-NI implies t-probing security. If an algorithm satisfies t-NI, then only t shares of each input are required to simulate the result, and the perfect sharing of the secret allows these shares to be simulated independently of the secret input itself. A significant advantage is that adapting maskVerif to verify this new security notion was straightforward. While t-NI is satisfied by the ISW multiplication gadget, it unfortunately still faces challenges with composition.

The issue arises with the multiplication $x * x$. While t-NI ensures that there exist sets I_1 and I_2 of input shares for the arguments that are sufficient to simulate t observations in the multiplication, with $|I_1| \leq t$ and $|I_2| \leq t$, the complication occurs when both arguments are the same. In this case, we need $I_1 \cup I_2$ shares of x to perform the simulation, and the cardinality of $I_1 \cup I_2$ can exceed t.

This problem was known, and the proposed solution for implementing such a

program was to *refresh* one of the inputs to the multiplication gadget. A refresh gadget aims to securely recompute a sharing of its input. The initial refresh gadget was proposed by Rivain and Prouff [RP10]. We discovered through maskVerif that while its composition with multiplication was insecure, it became secure when the refresh gadget was implemented using an ISW multiplication by a sharing of 1, i.e., the tuple $(1, 0, \ldots, 0)$.

So we have tried to understand which property verify the ISW multiplication and not the initial refresh gadget of Rivain and Prouff. We observed that the ISW multiplication gadget satisfies a useful property, which we have termed *strong non-interference* (t-SNI):

To simulate n_1 internal observations and n_2 observations from the output shares, only n_1 shares of each inputs are needed, as long as $n_1 + n_2 \leq t$.

Now consider $x * \text{refresh}(x)$, where both the multiplication and the refresh are assumed to be t-SNI. Assume that n_1 observations are made during the multiplication and n_2 during the refresh, with the total number of observations bounded by t, i.e., $n_1 + n_2 \leq t$.

To simulate the observations made during the multiplication, we only need n_1 shares from the first occurrence of x and n_1 shares from the output of $\text{refresh}(x)$. Since the refresh gadget is SNI, only n_2 input shares of x are required to simulate the necessary n_1 outputs of $\text{refresh}(x)$ and its n_2 internal observations. In the end, all observations can be simulated using only $n_1 + n_2$ shares.

In [36], we introduced the notions of Non-Interference (NI) and Strong Non-Interference (SNI), along with a type system that ensures the correct composition of gadgets. This type system tracks the minimal number of shares needed to simulate any t-tuple of observations by using symbolic sets and constraints on their cardinality. It take into account that addition gadget follow a particular property due to the fact that they are linear. The type system guarantees that whenever a multiplication gadget is encountered, the number of internal observations and output shares required to simulate the observations made in the gadget's continuation is less than t. It also ensures that the composition of all gadgets requires fewer than t shares of each input, which ensure NI and so the t-probing security.

Additionally, the system can automatically patch an implementation by adding refresh gadgets when necessary. This led to the development of the compiler maskComp [maskComp], which takes a C implementation and automatically transforms it into a masked implementation, replacing additions and multiplications with the corresponding gadgets and adding refresh gadgets where required.

Finally, I extended the maskVerif tool to support the notions of Non-Interference (NI) and Strong Non-Interference (SNI), enabling it to account for stronger leakage models (closer to the hardware) that consider transitions and glitches. We leveraged

maskVerif to verify the security of new multiplication and refresh gadgets, which reduce the amount of required randomness.

4.5 Main contribution

In [27, 15, 10, 7], we provide methods for guaranteeing the preservation of constant time by a compiler, we also provide different techniques to ensure that a program is CT, either based on type systems or via program equivalence using the EasyCrypt proof assistant. In [22, 6] we provide different techniques to protect against Spectre attacks.

We established a strong connection between the notions of security used by the masking community and the notion of probabilistic non-interference in the programming language community. This connection made it possible to establish new theoretical foundations for masking and to break through two technological barriers. We developed an automatic algorithm for checking the probabilistic non-interference of a masked implementation. This algorithm verifies the security of an implementation against a particular attack strategy. However, the security notions used in masking require the implementation to be protected against all the attacker's strategies. This number of strategies is exponential depending on the size of the program and the desired level of protection (up to billions of cases for a multiplication algorithm). We developed new algorithms to factorise the verification effort and make it practical for basic implementations. We then proposed the first correct method for analysing complete implementations. This method is based on a new security notion, called SNI (Strong Non Interference), which enables compositional reasoning. This notion of SNI has become the de facto standard for masked implementations. We then used this notion of SNI to develop the first compiler to generate guaranteed masked implementations for arbitrary levels of protection (maskComp).

Chapter 5

Perspectives

My overarching goal for the next few years is to promote the adoption of high-assurance cryptographic software, through foundational and applied work to (i) lower the entry bar for high-assurance methods; and (ii) keep the scope of the high-assurance approach aligned with the state-of-the-art in cryptographic research.

Jasmin Language and its compiler A key area for improvement is extending the Jasmin compiler to support more architectures, which is essential for broadening the deployment of cryptographic libraries. Over the last two years, we made substantial progress by adding support for the ARM-v7 backend. This required significant effort, as most compiler passes had to be rewritten to be parametric, accommodating architectural variations such as pointer sizes and low-level instruction constraints.

Building on this, our next priority is to support two additional architectures: RISC-V and ARM-v8. The rationale for ARM-v8 is clear, given its widespread use in servers like Amazon's cloud infrastructure. While RISC-V is less common, its open architecture offers an exciting opportunity for future research and development. Integrating RISC-V into Jasmin not only opens new research avenues but also positions us to take advantage of its potential growth in the industry.

In parallel, we should eliminate some unnecessary restrictions in the Jasmin language. For instance, the current calling convention for exported functions limits them to a maximum of five arguments, which hampers the implementation of certain interfaces and the integration of Jasmin code into existing libraries. Another limitation is the requirement that array sizes must be statically known at compile time. Currently, the work around is to generate multiple instances of the same function, thereby increasing code size. This restriction originates from the stack-allocation algorithm used in Jasmin.

The stack-allocation algorithm is particularly complex because it must ensure that persistent arrays, (which are convenient for proofs and program analysis) can

be replaced by in-place arrays (which are efficient). Improving this algorithm would lead to significant advancements in the language. Allowing to improve the efficiancy and to reduce the size of the code of the post-quantum cryptographic library libjade. It would also be valuable to explore whether this challenge is related to Rust's ownership model, which guarantees that every value has a single owner, as there may be similar underlying concepts.

Ensuring (speculative) constant time in the post-quantum setting On one front, I aim to extend our work on preserving constant-time to speculative constant-time (SCT). One motivation of this work is theoretical, the other is practical. It requires us to develop a leakage model at the source level that is strong enough to ensure the preservation of SCT, allowing us to adapte our type system for SCT to fit the leakage model. Having the type system at the source level is important from the user perspective, since reporting of error is much more accurate.

On an other front, I think it is not reasonnable to maintain the proof of preservation of CT in a realistic and evolving compiler like Jasmin. Each new compilation pass requires a new proof that the pass preserves constant-time execution. The situation becomes even more complex with speculative constant-time, as speculative semantics must be incorporated at the source level and to all intermediate semantics used during compilation. I plan to follow a more pragmatique approach consisting of devellopping a type system (for CT and SCT) at the assembly level. For that the main difficulty is to a solid pointer analysis at the assembly level. I believe that the analysis can be provided by the Jasmin stack allocation pass of the compiler, then we can proof preservation of the analysis down to assembly.

Another open question involves justifying the use of the *declassify* construct, which is often used in type systems for constant-time to bypass their incompleteness. *declassify* are necessary to type check the random sampling algorithm used in Kyber and Dilithium or Falcom, since this algorithm branch on secret dependant datas. However, the core of their security proof demonstrates that rejection sampling (and consequently the number of loop iterations) is probabilistic independent of the secret. I think we should introduce the new security notion of approximate probabilistic constant-time, meaning that an adversary has a negligible probability of distinguishing two distribution of leakage.

Ideally we should perform the security proof in a model where the adversary has access to the full leakage. Unfortunatelly, this will pollute a lot the security proofs with detail related to the notion of constant time. I think it is possible to restrict the leakage to the declassified values, by providing a proof that the full leakage is in fact a deterministic function of the declassifier values (and the public part of the memory).

Provable security in the QROM Ongoing efforts to formally verify the security of NIST PQC finalist schemes in EasyCrypt show that it is possible and generally beneficial to eschew quantum reasoning using carefully selected QROM lemmas. This is the approch we have followed in [12]. However, this approach has shortcomings. First, several QROM proof tools, for instance (some variants of) the one-way to hiding lemma and the compressed oracle techniques, expose quantum computations. Second, these QROM lemmas have multiple variants, all of which form part of the Trusted Computing Base. This creates the risk of building formally verified proofs based on incorrect axioms. We plan to overcome these limitations by strengthening the foundations of EasyCrypt, and developing libraries for all QROM techniques. In the first case, we intend to develop better relational Hoare logics for quantum computations. In the second case, we intend to justify QROM lemmas formally within a general-purpose proof assistant, for instance using the denotational semantics of quantum programs provided by the **CoqQ** framework[ZBS+23] or the **qrhl** [Unr19] tool.

List of my publications 2002-2023

[1] Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, and Peter Schwabe. High-assurance zeroization. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):375–397, 2024.

[2] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality tests, automatically: Deriving eqtype instances for jasmin's data types with coq-elpi. In Robbert Krebbers, Dmitriy Traytel, Brigitte Pientka, and Steve Zdancewic, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*, pages 167–181. ACM, 2023.

[3] 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. Formally verifying kyber episode IV: implementation correctness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(3):164–193, 2023.

[4] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. Fixing and mechanizing the security proof of fiat-shamir with aborts and dilithium. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part V*, volume 14085 of *Lecture Notes in Computer Science*, pages 358–389. Springer, 2023.

[5] Manuel Barbosa, François Dupressoir, Benjamin Grégoire, Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. Machine-checked security for rmxmss as in RFC 8391 and $\mathrm {SPHINCSˆ{+}} $. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara,*

CA, USA, August 20-24, 2023, Proceedings, Part V, volume 14085 of *Lecture Notes in Computer Science*, pages 421–454. Springer, 2023.

[6] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 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*, pages 1094–1111. IEEE, 2023.

[7] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing fine-grained constant-time policies. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 83–96. ACM, 2022.

[8] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[9] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in fine-grained leakage models: Construction, implementation and verification. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):189–228, 2021.

[10] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Structured leakage and applications to cryptographic constant-time and cost. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 462–476. ACM, 2021.

[11] Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, and Pierre-Yves Strub. Mechanized proofs of adversarial complexity and application to universal composability. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2541–2563. ACM, 2021.

[12] Manuel Barbosa, Gilles Barthe, Xiong Fan, Benjamin Grégoire, Shih-Han Hung, Jonathan Katz, Pierre-Yves Strub, Xiaodi Wu, and Li Zhou. Easypqc: Verifying post-quantum cryptography. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on*

Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021, pages 2564–2586. ACM, 2021.

[13] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 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*, pages 1884–1901. IEEE, 2021.

[14] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Improved parallel mask refreshing algorithms: generic solutions with parametrized non-interference and automated optimizations. *J. Cryptogr. Eng.*, 10(1):17–26, 2020.

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

[16] Mohamad El Laz, Benjamin Grégoire, and Tamara Rezk. Security analysis of elgamal implementations. In Pierangela Samarati, Sabrina De Capitani di Vimercati, Mohammad S. Obaidat, and Jalel Ben-Othman, editors, *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications, ICETE 2020 - Volume 2: SECRYPT, Lieusaint, Paris, France, July 8-10, 2020*, pages 310–321. ScitePress, 2020.

[17] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 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*, pages 965–982. IEEE, 2020.

[18] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Matthew Campagna, Ernie Cohen, Benjamin Grégoire, Vitor Pereira, Bernardo Portela, Pierre-Yves Strub, and Serdar Tasiran. A machine-checked proof of security for AWS key management service. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 63–78. ACM, 2019.

[19] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. Machine-checked proofs for cryptographic

standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 1607–1622. ACM, 2019.

[20] Gilles Barthe, Benjamin Grégoire, Charlie Jacomme, Steve Kremer, and Pierre-Yves Strub. Symbolic methods in computational cryptography proofs. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 136–151. IEEE, 2019.

[21] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[22] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a DSL for timing-sensitive computation. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 174–189. ACM, 2019.

[23] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.*, 2(POPL):57:1–57:29, 2018.

[24] Gilles Barthe, Xiong Fan, Joshua Gancher, Benjamin Grégoire, Charlie Jacomme, and Elaine Shi. Symbolic proofs for lattice-based cryptography. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 538–555. ACM, 2018.

[25] Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. Vectorizing higher-order masking. In Junfeng Fan and Benedikt Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2018.

[26] Cécile Baritel-Ruet, François Dupressoir, Pierre-Alain Fouque, and Benjamin Grégoire. Formal security proof of CMAC and its variants. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 91–104. IEEE Computer Society, 2018.

[27] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 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*, pages 328–343. IEEE Computer Society, 2018.

[28] Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. An assertion-based program logic for probabilistic programs. In Amal Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 117–144. Springer, 2018.

[29] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, 2018.

[30] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1807–1823. ACM, 2017.

[31] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1989–2006. ACM, 2017.

[32] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.

[33] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving uniformity and independence by self-composition and coupling. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 385–403. EasyChair, 2017.

[34] Gilles Barthe, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Coupling proofs are probabilistic product programs. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 161–174. ACM, 2017.

[35] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 55–67. ACM, 2016.

[36] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.

[37] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. A program logic for union bounds. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 107:1–107:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[38] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In Martin Grohe, Eric Koskinen, and Natarajan Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 749–758. ACM, 2016.

[39] Gilles Barthe, Benjamin Grégoire, and Benedikt Schmidt. Automated proofs of pairing-based cryptography. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 1156–1168. ACM, 2015.

[40] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[41] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. Relational reasoning via probabilistic coupling. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2015.

[42] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. Synthesis of fault attacks on cryptographic implementations. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1016–1027. ACM, 2014.

[43] Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Mehdi Tibouchi, and Jean-Christophe Zapalowicz. Making RSA-PSS provably secure against non-random faults. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2014.

[44] Joseph A. Akinyele, Gilles Barthe, Benjamin Grégoire, Benedikt Schmidt, and Pierre-Yves Strub. Certified synthesis of efficient batch verifiers. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 153–165. IEEE Computer Society, 2014.

[45] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 193–206. ACM, 2014.

[46] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella Béguelin. Verified indifferentiable hashing into elliptic curves. *J. Comput. Secur.*, 21(6):881–917, 2013.

[47] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1247–1260. ACM, 2013.

[48] Gilles Barthe, George Danezis, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Verified computational differential privacy with applications to smart metering. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 287–301. IEEE Computer Society, 2013.

[49] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In Alessandro Aldini, Javier López, and Fabio Martinelli, editors, *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[50] Gilles Barthe, Benjamin Grégoire, César Kunz, Yassine Lakhnech, and Santiago Zanella Béguelin. Automation in computer-aided cryptography: Proofs, attacks and designs. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 7–8. Springer, 2012.

[51] Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. Verified security of merkle-damgård. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 354–368. IEEE Computer Society, 2012.

[52] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Computer-aided cryptographic proofs. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 11–27. Springer, 2012.

[53] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In Jeremy Gibbons and Pablo Nogueira, editors, *Mathematics of Program Construction - 11th International Conference, MPC 2012, Madrid, Spain, June 25-27, 2012. Proceedings*, volume 7342 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2012.

[54] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Federico Olmedo, and Santiago Zanella Béguelin. Verified indifferentiable hashing into elliptic curves. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, volume 7215 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2012.

[55] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Computer-aided cryptographic proofs. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 1–2. Springer, 2012.

[56] Benjamin Grégoire. Recent advances in the formal verification of cryptographic systems: Turing's legacy. *ERCIM News*, 2012(91), 2012.

[57] Jan Olaf Blech and Benjamin Grégoire. Certifying compilers using higher-order theorem provers as certificate checkers. *Formal Methods Syst. Des.*, 38(1):33–61, 2011.

[58] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers

to coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[59] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer, 2011.

[60] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, 2011.

[61] Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security verifiable IND-CCA security of OAEP. In Aggelos Kiayias, editor, *Topics in Cryptology - CT-RSA 2011 - The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, volume 6558 of *Lecture Notes in Computer Science*, pages 180–196. Springer, 2011.

[62] Gilles Barthe, Daniel Hedin, Santiago Zanella Béguelin, Benjamin Grégoire, and Sylvain Heraud. A machine-checked formalization of sigma-protocols. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 246–260. IEEE Computer Society, 2010.

[63] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to SAT verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010.

[64] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Programming language techniques for cryptographic proofs. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2010.

[65] Benjamin Grégoire and Jorge Luis Sacchini. On strong normalization of the calculus of constructions with type-based termination. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2010.

[66] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. *ACM Trans. Program. Lang. Syst.*, 31(5):18:1–18:45, 2009.

[67] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, César Kunz, and Anne Pacalet. Implementing a direct method for certificate translation. In Karin K. Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 541–560. Springer, 2009.

[68] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101. ACM, 2009.

[69] Santiago Zanella Béguelin, Gilles Barthe, Benjamin Grégoire, and Federico Olmedo. Formally certifying the security of digital signature schemes. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 237–250. IEEE Computer Society, 2009.

[70] Benjamin Grégoire, Loïc Pottier, and Laurent Théry. Proof certificates for algebra and their application to automatic geometry theorem proving. In Thomas Sturm and Christoph Zengler, editors, *Automated Deduction in Geometry - 7th International Workshop, ADG 2008, Shanghai, China, September 22-24, 2008. Revised Papers*, volume 6301 of *Lecture Notes in Computer Science*, pages 42–59. Springer, 2008.

[71] Gilles Barthe, Benjamin Grégoire, and Mariela Pavlova. Preservation of proof obligations from java to the java virtual machine. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 83–99. Springer, 2008.

[72] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In Michael Kaminski and Simone Martini, editors, *Computer Science Logic, 22nd International Workshop, CSL 2008, 17th Annual Conference of the EACSL, Bertinoro, Italy, September 16-19, 2008. Proceedings*, volume 5213 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2008.

[73] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Formal certification of elgamal encryption. In Pierpaolo Degano, Joshua D. Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Malaga, Spain, October 9-10, 2008, Revised Selected Papers*, volume 5491 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2008.

[74] Gilles Barthe, Benjamin Grégoire, and Colin Riba. A tutorial on type-based termination. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 100–152. Springer, 2008.

[75] Bruno Barras, Pierre Corbineau, Benjamin Grégoire, Hugo Herbelin, and Jorge Luis Sacchini. A new elimination rule for the calculus of inductive constructions. In Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro, editors, *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2008.

[76] Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas P. Jensen, and David Pichardie. The MOBIUS proof carrying code infrastructure. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 6th International Symposium, FMCO 2007, Amsterdam, The Netherlands, October 24-26, 2007, Revised Lectures*, volume 5382 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2007.

[77] Benjamin Grégoire and Jorge Luis Sacchini. Combining a verification condition generator for a bytecode language with static analyses. In Gilles Barthe and Cédric Fournet, editors, *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*, volume 4912 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.

[78] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2006.

[79] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to pocklington certificates in type theory. In Masami Hagiya and Philip Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2006.

[80] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. JACK - A tool for validation of security and behaviour of java applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2006.

[81] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Cic[ˆ( )]: type-based termination of recursive definitions in the calculus of inductive constructions. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2006.

[82] Gilles Barthe, Benjamin Grégoire, César Kunz, and Tamara Rezk. Certificate translation for optimizing compilers. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 301–317. Springer, 2006.

[83] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. MOBIUS: mobility, ubiquity, security. In Ugo Montanari, Donald Sannella, and Roberto Bruni, editors, *Trustworthy Global Computing, Second Symposium, TGC 2006, Lucca, Italy, November 7-9, 2006, Revised Selected Papers*, volume 4661 of *Lecture Notes in Computer Science*, pages 10–29. Springer, 2006.

[84] Gilles Barthe, Benjamin Grégoire, Marieke Huisman, and Jean-Louis Lanet, editors. *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Second International Workshop, CASSIS 2005, Nice, France, March 8-11, 2005, Revised Selected Papers*, volume 3956 of *Lecture Notes in Computer Science*. Springer, 2006.

[85] Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In C.-H. Luke Ong, editor, *Computer Science Logic, 19th International Workshop, CSL 2005, 14th Annual Conference of the EACSL, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3634 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2005.

[86] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. Practical inference for type-based termination in a polymorphic setting. In Pawel Urzyczyn, editor, *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2005.

[87] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in coq. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2005.

[88] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2004.

[89] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 235–246. ACM, 2002.

Software related to my research 2002-2023

[bignums] A coq library provides bign, bigz, and bigq that used to be part of the standard library. https://github.com/coq-community/bignums.

[CertiCrypt] A framework that enables the machine-checked construction and verification of code-based proofs in coq. https://github.com/EasyCrypt/certicrypt.

[coqprime] Primality proofs using pocklington certificate and elliptic curve certificate. https://github.com/thery/coqprime.

[EasyCrypt] Computer-aided cryptographic proofs. https://github.com/EasyCrypt/easycrypt.

[Jasmin] A language and a compiler designed for writing high-assurance and high-speed cryptography. https://github.com/jasmin-lang/jasmin.

[maskComp] A compiler for automatic generation of masked implementation. https://sites.google.com/site/maskingcompiler/home.

[maskVerif] A automatic checker for verifying masked implementation. https://gitlab.com/benjgregoire/maskverif.

[Ring/Field] The ring and field Coq tactics. https://coq.inria.fr/refman/addendum/ring.html.

[vm_compute] The native/vm_compute mecanism/tactic. https://coq.inria.fr/refman/proofs/writing-proofs/equality.html#coq:tacn.vm_compute.

[ZooCrypt] Automatic proofs and synthesis of cryptographic primitives. https://github.com/ZooCrypt/AutoGnP.

Bibliography

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 104–115. ACM, 2001.

[AP13] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy, SP 2013*, pages 526–540. IEEE Computer Society, 2013.

[BBPV12] Billy Bob Brumley, Manuel Barbosa, Dan Page, and Frederik Vercauteren. Practical realisation and elimination of an ecc-related software bug attack. In Orr Dunkelman, editor, *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings*, volume 7178 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2012.

[BCJP09] Frédéric Besson, David Cachera, Thomas P. Jensen, and David Pichardie. Certified static analysis by abstract interpretation. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 223–257. Springer, 2009.

[BDF+11] Dan Boneh, Özgür Dagdelen, Marc Fischlin, Anja Lehmann, Christian Schaffner, and Mark Zhandry. Random oracles in a quantum world. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 41–69. Springer, 2011.

[Ber] Dan Bernstein. Writing high-speed software.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes, 2005. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310.

[BHK09] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. Subtleties in the definition of IND-CCA: when and how should challenge-decryption be disallowed? *IACR Cryptol. ePrint Arch.*, page 418, 2009.

[BHK15] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. Subtleties in the definition of IND-CCA: when and how should challenge decryption be disallowed? *J. Cryptol.*, 28(1):29–48, 2015.

[BJPT10] Frédéric Besson, Thomas P. Jensen, David Pichardie, and Tiphaine Turpin. Certified result checking for polyhedral analysis of bytecode programs. In Martin Wirsing, Martin Hofmann, and Axel Rauschmayer, editors, *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24-26, 2010, Revised Selected Papers*, volume 6084 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2010.

[BKR94] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of cipher block chaining. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 1994.

[BMS+20] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 54–72. IEEE, 2020.

[BMW+19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Breaking virtual memory protection and the SGX ecosystem with foreshadow. *IEEE Micro*, 39(3):66–74, 2019.

[Bol09] Alexandra Boldyreva. Strengthening security of RSA-OAEP. In Marc Fischlin, editor, *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings*, volume 5473 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2009.

[BPB12] Gilles Barthe, David Pointcheval, and Santiago Zanella Béguelin. Verified security of redundancy-free encryption from rabin and RSA. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 724–735. ACM, 2012.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with RSA and rabin. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 1996.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *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*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, 2006.

[BZ13] Dan Boneh and Mark Zhandry. Quantum-secure message authentication codes. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 592–608. Springer, 2013.

[Car] Chandler Carruth. Speculative load hardening – a Spectre variant #1 mitigation technique. LLVM documentation. https://llvm.org/docs/SpeculativeLoadHardening.html.

[DPS15] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. 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*, pages 73–87. IEEE Computer Society, 2015.

[FMP13] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 345–365. Springer, 2013.

[FOPS04] Eiichiro Fujisaki, Tatsuaki Okamoto, David Pointcheval, and Jacques Stern. RSA-OAEP is secure under the RSA assumption. *J. Cryptol.*, 17(2):81–104, 2004.

[GK13] Shay Gueron and Vlad Krasnov. The fragility of AES-GCM authentication algorithm. Cryptology ePrint Archive, Report 2013/157, 2013. http://eprint.iacr.org/2013/157.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 251–261. Springer, 2001.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 197–206. ACM, 2008.

[GSE20] Mordechai Guri, Yosef A. Solewicz, and Yuval Elovici. Fansmitter: Acoustic data exfiltration from air-gapped computers via fans noise. *Comput. Secur.*, 91:101721, 2020.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded

Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[JSS+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KPVV16] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC 2015. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 573–582, 2016.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[Lin02] Torgny Lindvall. *Lectures on the coupling method*. Courier Corporation, 2002.

[Poi05] David Pointcheval. OAEP: optimal asymmetric encryption padding. In Henk C. A. van Tilborg, editor, *Encyclopedia of Cryptography and Security*. Springer, 2005.

[Pot01] François Pottier. Simplifying subtyping constraints: A theory. *Inf. Comput.*, 170(2):153–183, 2001.

[RBBG21] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1451–1468. USENIX Association, 2021.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[RMR+21] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1852–1867. IEEE, 2021.

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016.

[Sho01] Victor Shoup. OAEP reconsidered. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology*

Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings, volume 2139 of *Lecture Notes in Computer Science*, pages 239–259. Springer, 2001.

[SSL+19] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2019.

[Unr15] Dominique Unruh. Revocable quantum timed-release encryption. *J. ACM*, 62(6):49:1–49:76, 2015.

[Unr19] Dominique Unruh. Quantum relational hoare logic. *Proc. ACM Program. Lang.*, 3(POPL):33:1–33:31, 2019.

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2/3):167–188, 1996.

[XZH+19] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq (work in progress). *CoRR*, abs/1906.00046, 2019.

[ZBS+23] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. Coqq: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL):833–865, 2023.

[Zha12] Mark Zhandry. Secure identity-based encryption in the quantum random oracle model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 758–775. Springer, 2012.