Cryptographic Simulator Synthesis Using Program Logics

Toward a Framework for Mechanizing Cryptographic Reductions

Adrien Koutsos
Inria Paris

Aymeric Fromherz
Inria Paris

September 2025

Location: The internship will take place at Inria Paris, in the Prosecco team.

Contact information:

- Adrien Koutsos: adrien.koutsos@inria.fr
- Aymeric Fromherz: aymeric.fromherz@inria.fr

Expected abilities of the student. The student will need a strong background in logics, proof theory and program verification. Knowledge in security and cryptography is a plus, but is *not* required: the necessary background will be acquired during the internship if needed.

Computer-aided cryptography. Cryptography is vital to protect communications: for example, the TLS protocol ensures the security of all HTTPS communication. Unfortunately, cryptographic designs are routinely found flawed (e.g. [9, 1], to cite but a few TLS attacks). Formal verification can be used to obtain strong guarantees on the security of cryptographic designs, by formally proving their security. Computer-aided verification, where the proof of security itself is mechanized and verified by a dedicated tool, provides the highest level of guarantees.

The field of computer-aided cryptographic verification is now established, with several tool being actively developed and used (e.g. EasyCrypt [11] and Squirrel [12]). Verifying cryptographic programs requires to deal with a number of aspects, including concurrency (protocols are intrinsically asynchronous), probabilities (randomness is pervasive in cryptography), and complexity analysis (bounding the run-times of adversary is needed, e.g. to guard against brute-force attacks). Further, a verification framework with higher-order features are desirable, to allow for proof-reuse

and modular reasoning. The combination of all these features makes computer-aided cryptography a challenging sub-field of program verification.

The security of a cryptographic design is often expressed using *games*. A game $\mathcal{G} = (\mathcal{G}_0, \mathcal{G}_1)$ is a pair of programs: typically, $\mathcal{G}_0$ could represent the execution of a protocol, and $\mathcal{G}_1$ may be an idealized version of the protocol where security is obvious by construction (e.g. $\mathcal{G}_1$ could be the target protocol, except that all messages exchanged over the network have been replaced by zeroes, ensuring their confidentiality). A game is secure iff. no adversary (formally, a polynomial-time probabilistic program) can distinguish between them, except with negligible probability[1]:

$$\forall \mathcal{A} : \text{PTIME}.\ |\Pr(\mathcal{A}(\mathcal{G}_0) = 1) - \Pr(\mathcal{A}(\mathcal{G}_1) = 1)| \leq \epsilon_{\mathsf{negl.}} \tag{1}$$

The goal of a cryptographic proof is to formally establish such probability bounds. This kind of proofs are complex, and mechanizing them requires significant manual proof efforts (proofs are regularly thousands of lines long, e.g. the security proof for SHA3 of [3] is around 17 kLoC). To tackle this issue, it is useful to design logics which are as elegant and usable as possible, reducing the proof-burden put on users. As an example from program verification, separation logics allow to facilitate reasoning on the heap in a way that is modular and amenable to automation. We have similar aims, but for the verification of cryptographic proofs.

Cryptographic reductions. A prime candidate for this are *cryptographic reductions*: assuming the security of some hardness game $\mathcal{H} = (\mathcal{H}_0, \mathcal{H}_1)$, we can prove that another game $\mathcal{G}$ is secure by exhibiting an adversary against $\mathcal{H}$ that can simulate $\mathcal{G}$; indeed, an adversary for $\mathcal{G}$ composed with that simulator would yield an adversary for $\mathcal{H}$. That is, to reduce $\mathcal{G}$ to $\mathcal{H}$, we must exhibit a single *simulator* $\mathcal{S}$ such that, roughly:

(proba.) $\mathcal{S}(\mathcal{H}_0) = \mathcal{G}_0$ and $\mathcal{S}(\mathcal{H}_1) = \mathcal{G}_1$ **(complexity)** $\mathcal{S}$ is PTIME.

Writing simulators in detail is tedious and error prone, involving a lot of boilerplate code for a few interesting steps: we want an approach reducing the necessary user inputs to a minimum. More precisely, we want dedicated logics that can *synthesize* correct cryptographic simulators.

[1] Very roughly, negligible means exponentially small in the security parameter η (typically, η is the length of the cryptographic keys).

State-of-the-art and limitations. In [6], a logic has been proposed to do exactly that: the logic features a judgement $\#(\vec{h}_0; \vec{h}_1) \triangleright \#(g_0; g_1)$ called *bi-deduction*, which essentially states that there exists a simulator $\mathcal{S}$ such that $\mathcal{S}(\vec{h}_i) = g_i$ for any $i \in \{0, 1\}$. Then, [6] proposed a proof-system for bi-deduction, which serves as basis of an automated proof-search procedure. But this logic and associated proof-system suffer from several limitations and drawbacks:

- *Non-standard.* The logic formulation is non-standard, limiting its adoptions. In particular, the target program $\mathcal{G}_i$ and the simulated execution $\mathcal{S}(\mathcal{H}_i)$ are usually not equal, but only need to yield identical *probabilistic distributions* for their outputs. The logic of [6] does this by indirectly establishing the existence of a *probabilistic coupling* [7] through so-called *name constraints*. A more direct approach would be more intuitive and thus desirable.

- *Complexity.* The proof-system only supports target program (g_0, g_1) with bounded loops of the form **for** $i = 0$ **to** N, where N is a constant independent from the security parameter η. This restricts the logic to proving parametric security [5], which is weaker than the polynomial security as stated in Equ. (1).

- *Approximated simulation.* The bi-deduction $\triangleright$ requires that the simulator $\mathcal{S}$ *exactly* computes the target program. This could be weakened by allowing for a negligible probability of error during simulation. But this must be done carefully, so as to avoid the error to increase by more than a negligible quantity. We note that the interaction of negligible errors and loops with a polynomial number of interactions is delicate [13].

- *Higher-order.* The games are restricted to first-order programs: it would be interesting to extend this to a higher-order setting.

Internship Goals. During this internship, we will aim to design a logic overcoming the limitations described above while operating under the following design constraints:

- *Implicit simulators.* The logic should allow to build simulators without making them explicit. A syntax-directed approach that exploits as much as possible the shape of the target programs to simulate seems particularly adapted.

- *Elegant.* The logic should be elegant and intuitive to use. To that end, a standard-looking *program logic*, e.g. taking the form of a probabilistic Relational Hoare Logic [8] (pRHL), seems desirable. Such a logic would be standard

only on the surface, e.g. we expect it to capture relations between four different programs (the two target games, and the two executions of the simulators being built), instead of the usual two of pRHL.

- *Usable.* The logic should be usable and modular: we believe that separation logic could be the way to go.
- *Amenable to proof automation.* Automation allows to reduce boilerplate by requiring user intervention only for the most intricate steps (e.g. loop invariants). As opposed to [6], we do not aim for full automation, which might limit the planned extensions.
- *Mechanizable.* As we target computer-aided cryptography, the logic should be *mechanizable* in existing proof assistants. This should not be an issue, as Hoare-style separation logics are well-suited for this. While this is the end-goal, we do not expect this internship to go all-the-way to mechanization.

Proposed organization of the internship. In order to simplify the task of the intern and reduce the risk of failure, we propose a gradual approach, attacking each limitations of the state-of-the-art in isolation, one after the other, while operating under the design constraints presented above.

◇ *Task 1).* The first task will be to design an initial bare-bone version of the logic taking the form of a Hoare-style separation program logic. This will draw the general shape of the logic, and serve as basis for the following steps.

◇ *Task 2).* The next task could be to extend this base logic with: i) either support for more advanced complexity reasoning (e.g. using time-credits [10, 4]); ii) or to allow for approximations during simulation (e.g. using error credits [14, 2]). In both cases, we would rely on separation logic predicates, explaining why the base logic should be a separation logic. Both sub-tasks can be tackled in any order.

◇ *Task 3).* Finally, it could be interesting to move to a higher-order setting, allowing for more modular reasoning. We expect this to bring additional difficulties, e.g. for higher-order complexity reasoning. Still, this should be possible [14], though it may be non-trivial.

It seems unlikely that all tasks could be completed during the internship: designing a logic which either support advanced complexity reasoning or approximated simulation — i.e. tasks 1) plus 2.i) or 2.ii) — would be a satisfactory outcome. Completing this program, and then moving toward mechanization, could possibly be done during follow-up work, e.g. as part of a PhD (for which funding is available).

References

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Béguelin, and Paul Zimmermann. Imperfect forward secrecy: How diffie-hellman fails in practice. In *CCS*, pages 5–17. ACM, 2015.

[2] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Error credits: Resourceful reasoning about error bounds for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 8(ICFP):284–316, 2024.

[3] 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 *CCS*, pages 1607–1622. ACM, 2019.

[4] Robert Atkey. Amortised resource analysis with separation logic. In *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010.

[5] David Baelde, Caroline Fontaine, Adrien Koutsos, Guillaume Scerri, and Théo Vignon. A probabilistic logic for concrete security. In *CSF*, pages 324–339. IEEE, 2024.

[6] David Baelde, Adrien Koutsos, and Justine Sauvage. Foundations for cryptographic reductions in CCSA logics. In *CCS*, pages 2814–2828. ACM, 2024.

[7] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Formal certification of code-based cryptographic proofs. In *POPL*, pages 90–101. ACM, 2009.

[8] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In *MPC*, volume 7342 of *Lecture Notes in Computer Science*, pages 1–6. Springer, 2012.

[9] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Alfredo Pironti, and Pierre-Yves Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy*, pages 98–113. IEEE Computer Society, 2014.

[10] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *J. Autom. Reason.*, 62(3):331–365, 2019.

[11] The Easycrypt development team. The EasyCrypt Prover repository, accessed august 2025. https://github.com/EasyCrypt/easycrypt/.

[12] The Squirrel development team. The Squirrel Prover repository, accessed august 2025. https://github.com/squirrel-prover/squirrel-prover/.

[13] Marc Fischlin and Arno Mittelbach. An overview of the hybrid argument. *IACR Cryptol. ePrint Arch.*, page 88, 2021.

[14] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Approximate relational reasoning for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 9(POPL):1196–1226, 2025.