VTT

Beyond the Limits of PQC

Practical aspects of formal verification tools in cryptographic software implementations

Aada Illikainen, Valtteri Lipiäinen, Mikko Kiviharju

Introduction

Formal verification offers a complementary approach to traditional threat-based and empirical security techniques by mathematically proving security-relevant properties of specifications and implementations. It is especially important for security- and safety-critical software. A promising category of cybersecurity software suitable for verification are cryptographic implementations, and there is an increasing trend to formally verify cryptographic software, both specifications (e.g. the Signal protocol) and implementations (e.g. the HACL-library).

However, the label *"formally verified"* is often interpreted as *security-complete*, even though verification results depend on toolchains, assumptions, and how far the proofs extend in practice.

> Verified cryptographic software is a step forward, but its guarantees depend on the toolchains, trusted components, and how far the proof extends.

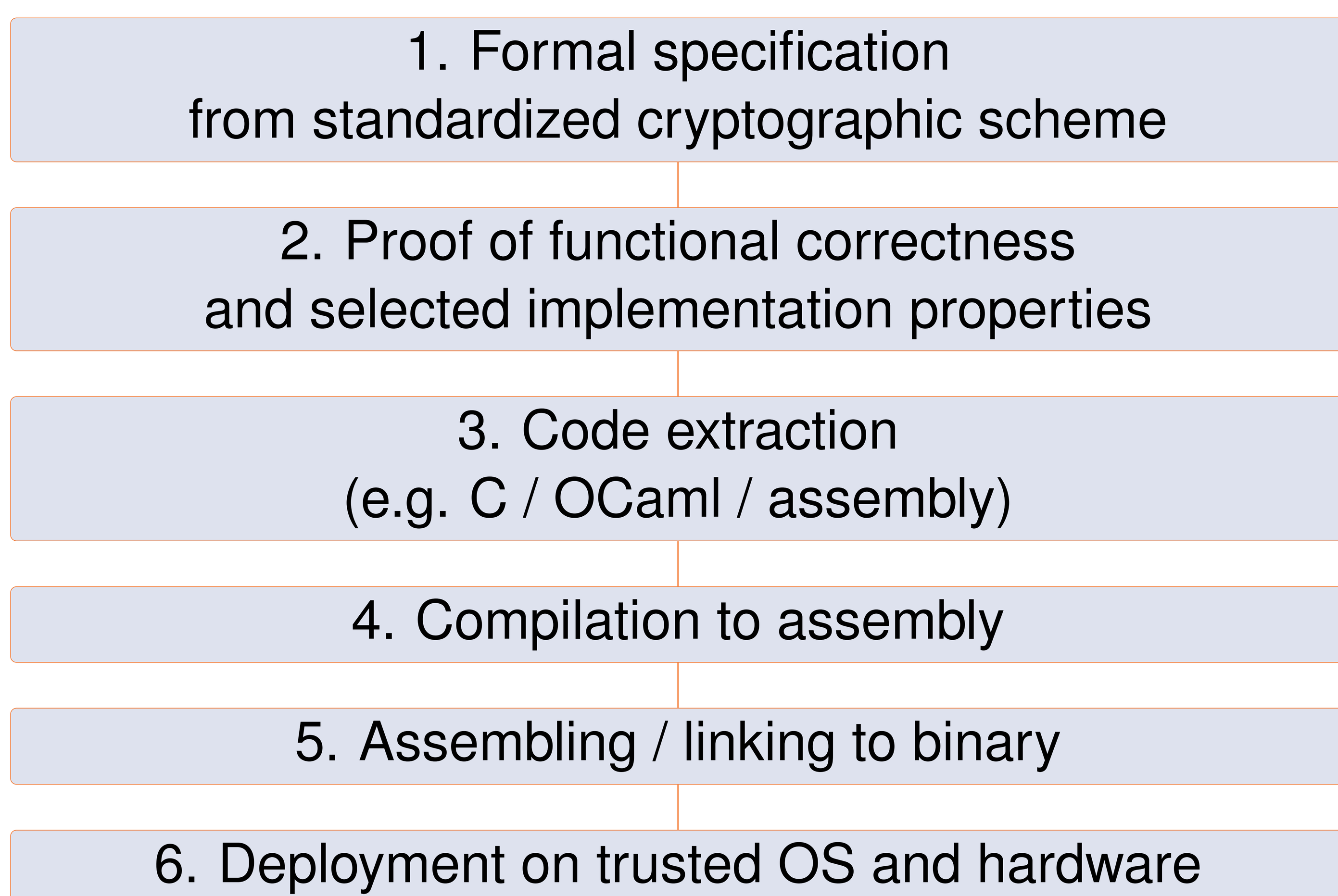
What this survey examines

- What properties do today's verification toolchains for cryptographic software actually prove?
- How are these tools used in documented, real-world cryptographic libraries?
- Where do assurance gaps remain from an end-user perspective?

Our focus is on **implementation-level verification**, not on protocol design proofs or full side-channel resistance.

Verification process

From the end-user's perspective, the assurance for the security claims should extend from the description of the cryptographic scheme down to the actual binary file executed on the target operating system and platform. However, threat vectors and proof techniques differ significantly throughout this path, and in practice there are different tools, frameworks and specification languages specialized at each software development lifecycle section.



Using formally verified tools in all stages helps preserving proved properties to the next stage.

Representative toolchains

Formal verification toolchains described in the chapter (see the following table) consist of theorem provers, their underlying SMT/SAT solvers, formal specification languages and other possible tools and features, such as translators from an implementation to a formal specification and compilers. These toolchains have been used in the formal verification effort for different cryptographic libraries that are described in the chapter.

Workbench	FSL	Low-level implementation	Theorem prover
F^*	(Pure) F^*	Low*, Vale	F^*
EasyCrypt	EasyCrypt	N/A	EasyCrypt
Jasmin	EasyCrypt	Jasmin	EasyCrypt
SAW	Cryptol	C, Java, Rust	Cryptol
Hax	Hacspec	Rust	F^*,Rocq Prover and others
Rocq	Gallina	N/A	Rocq Prover

List of verification tools used to formally verify cryptographic libraries introduced in the chapter.

BUSINESS FINLAND

This research received partial funding from the research project BLimPQC "Beyond the Limits of Post -Quantum Cryptography" funded by Business Finland, see `www.pqc.fi` *for further information.*

Case studies

The following table lists formally verified *cryptographic libraries*, i.e. software products that provide cryptographic functionality, most often to be used as a dependency for larger products. It presents a non-exhaustive list of formally verified cryptographic libraries identified in the literature.

Name	Functionality	Main tool	Main guarantee(s)
Libjade	PQC signatures & KEM	Jasmin	Cryptographic security
Libcrux	API for cryptographic functionality	Hacspec	Functional correctness
HACL*	NaCl API	F^*	Functional correctness
DICE*	DICE	F^*	Functional correctness
s2n	TLS	SAW	Functional correctness
ValeCrypt	NaCl API	Vale	Functional correctness
EverCrypt	NaCl API	F^*	Functional correctness
miTLS	TLS	F^*	Cryptographic security
DFP+ QUIC	QUIC encryption	F^*	Functional correctness
SBF+21 Plebeia	Merkle tree	F^*	Functional correctness
Bert13	TLS	Hax	Cryptographic security
Fiat cryptography	Elliptic curve operations	Rocq	Functional correctness

List of formally verified cryptographic libraries

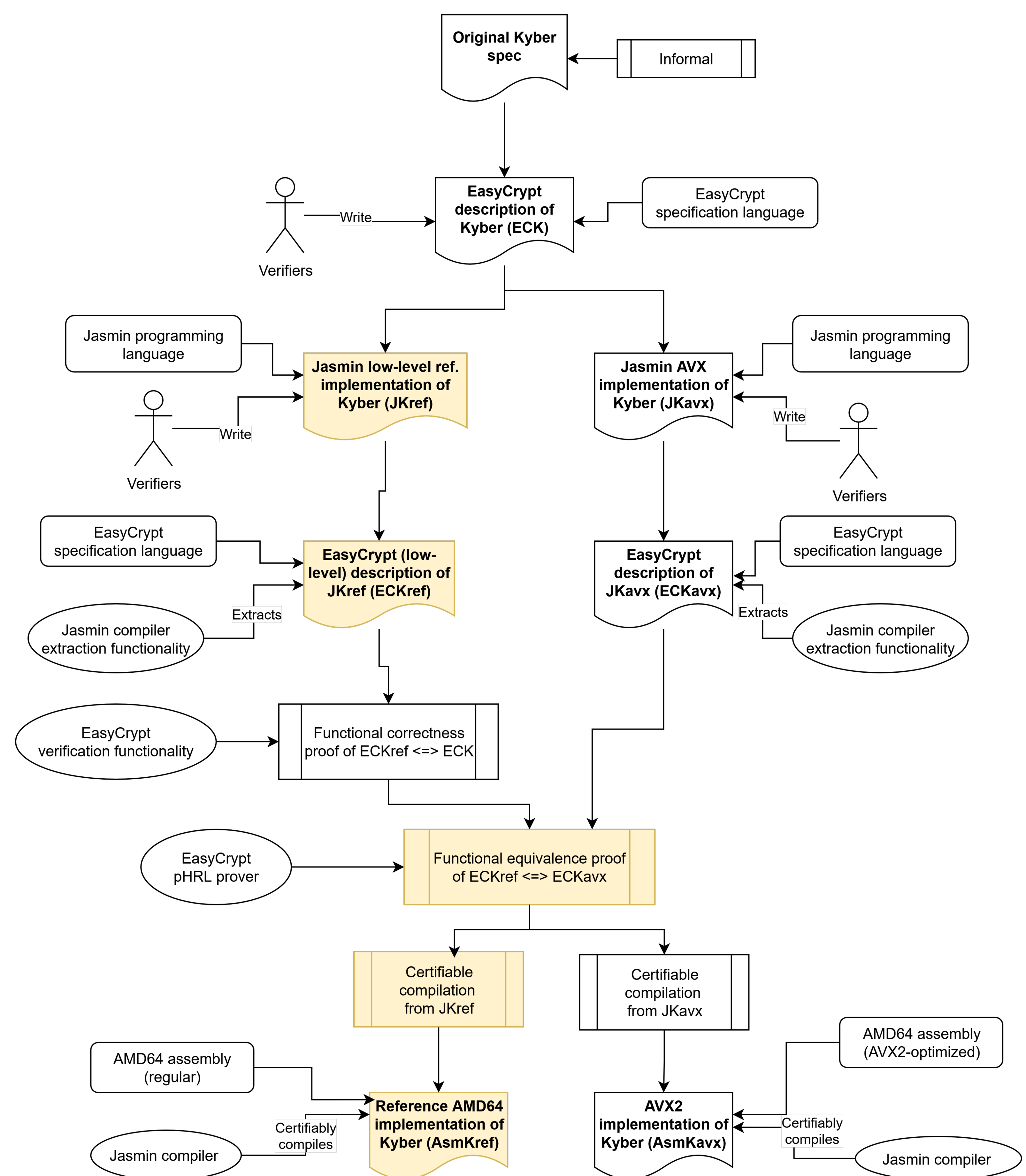


Figure from the chapter: example workflow used in ML-KEM verification.

Key lessons from the identified gaps

Tool maturity: Verifying modern cryptographic schemes, especially PQC, often requires extending existing verification tools and adding missing mathematical or implementation support.

Usability: Verification frameworks remain expert-level tools with steep learning curves. Limited automation, weak interactivity, and framework-specific proof support make verification workflows difficult to scale and maintain.

Security coverage: Functional correctness transfers guarantees across abstraction levels, but does not by itself create new implementation security guarantees.

Last-mile gap: In current toolchains, assurance often weakens after textual assembly. Compilers, assemblers, linkers, kernels, and hardware frequently remain inside the trusted computing base.

Practical implication: Verified cryptographic software is clearly more robust than unverified software, but it should not be treated as security-complete without examining assumptions, proof depth, and deployment context.

Contact: Aada Illikainen, Valtteri Lipiäinen, Mikko Kiviharju
aada.illikainen@vtt.fi, valtteri.lipiainen@vtt.fi, mikko.kiviharju@habilito.fi