Formal Security and Functional Verification of Cryptographic Protocol Implementations in Rust

Karthikeyan Bhargavan[1], Lasse Letager Hansen[2], Franziskus Kiefer[1], Jonas Schneider-Bensch[1], and Bas Spitters[2]

[1]Cryspen, Paris, France
[2]Aarhus University, Aarhus, Denmark

Abstract

We present an effective methodology for the formal verification of practical cryptographic protocol implementations written in Rust. Within a single proof framework, we show how to develop machine-checked proofs of diverse properties like runtime safety, parsing correctness, and cryptographic protocol security. All analysis tasks are driven by the software developer who writes annotations in the Rust source code and chooses a backend prover for each task, ranging from a generic proof assistant like F⋆ to dedicated crypto-oriented provers like ProVerif and SSProve. Our main contribution is a demonstration of this methodology on `Bert13`, a portable, post-quantum implementation of TLS 1.3 written in Rust and verified both for security and functional correctness. To our knowledge, this is the first security verification result for a protocol implementation written in Rust, and the first verified post-quantum TLS 1.3 library.

1 High-Assurance Cryptographic Protocols

The last decade has been a fertile time for the design and deployment of advanced cryptographic schemes and protocols, motivated by a variety of reasons ranging from the Snowden revelations to the popularity of cryptocurrencies. This trend promises to continue with new standards for post-quantum cryptography and new efforts around privacy-preserving machine learning, which will undoubtedly require novel protocol designs and fresh implementations.

Cryptographic protocol libraries, like OpenSSL[1], libsignal[2], and Bitcoin Core[3], have come to occupy an increasingly large part of the trusted computing base of modern computer systems, and are consequently held to a high standard. Any bug in these codebases is treated as a potentially costly vulnerability. Hence, the current period of rapid change raises concerns about the quality and security of all the new protocol code that is being developed and deployed.

In this work, we demonstrate a methodology for building high-assurance implementations of cryptographic protocols, where different core components can be formally verified for the desired security and correctness guarantees, using some of the most practical, state-of-the-art verification techniques available today.

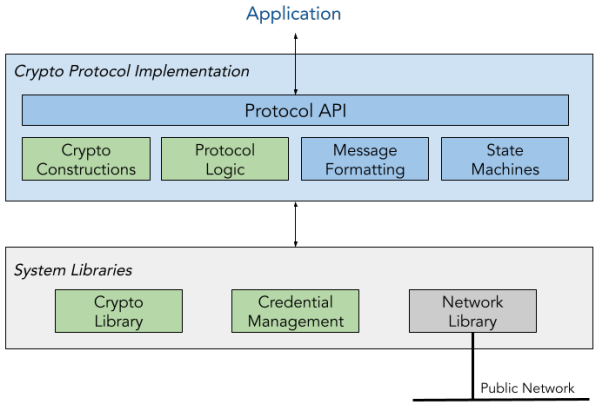


Figure 1: Crypto Protocol Implementation Components

Key Components of Protocol Implementations. Figure 1 depicts the high-level structure of a crypto protocol implementation.

The protocol relies on several system libraries: a *cryptographic library* that implements standard crypto algorithms; a *credential management library* that handles the retrieval, validation, and storage of long-term keys and credentials, such as X.509 certificates, private keys, and pre-shared keys; and a *networking library* that sends and receives messages over the untrusted network.

The protocol implementation itself consists of: protocol-specific *cryptographic constructions* that may compose multiple cryptographic algorithms; the core *protocol logic* that handles protocol message construction and processing; one or more *state machines* to keep track of protocol progress; and *message formatting* code to serialize and deserialize both public messages and internal cryptographic inputs. The protocol implementation combines these components to provide an API that can be used by the Application.

[1]`https://openssl-library.org/`
[2]`https://signal.org/docs/`
[3]`https://bitcoin.org/en/bitcoin-core/`

Bugs and Attacks. Each of these protocol components is security-critical and has a long history of attacks and vulnerabilities.

For example, consider implementations of the Transport Layer Security (TLS) protocol [54]. Prior works have found attacks on the specialized cryptographic constructions implemented in TLS [3, 7], allowing attackers to decrypt application messages. Other works have found flaws in the design and implementation of the protocol logic [15, 2], which weakened the expected authentication or confidentiality properties. Devastating state machine bugs found in TLS implementations allowed for all the protocol guarantees to be bypassed [12]. Ambiguities in the TLS message formats resulted in attacks on the authentication guarantees of the protocol [48, 19]. Of course, bugs are also frequently found in the libraries TLS depends on, e.g. in X.509 validation [36], and in the crypto library [35].

This wide variety of bugs and attacks is not restricted to standard protocols like TLS. Recent papers have found such attacks also on modern implementations of secure messengers [50], encrypted cloud storage [8], and multi-party computation [47]. Consequently, a methodology for developing high-assurance cryptographic protocol designs and implementations is an urgent necessity.

Formally Verification of Protocol Components. A growing field of research, sometimes called Computer-Aided Cryptography [9], is concerned with the formal analysis of and machine-checked proofs for the design and implementation of cryptographic mechanisms and protocols. Many of the tools and techniques developed in this field can be used to verify protocol components.

Domain-specific software verification tools have been developed to analyze the correctness and security of message formatting code [53, 61], the security of cryptographic constructions [10, 32], the formal analysis of protocol logic and state machines [16, 11, 42], and the formal verification of entire protocol implementations up to high-level APIs [17, 40]. A separate line of work has focused on developing formally verified cryptographic libraries in C and assembly (see e.g. [62, 30, 52, 5]). Recent work has also addressed verified implementations of X.509 public key certificate validation [25].

A common feature of most of these tools is that they address implementations written either in domain-specific languages (DSLs) or in highly-stylized subsets of mainstream languages. Consequently, these results are mainly applied to verification-oriented research code and do not consider idiomatic implementations written by protocol developers in C or Rust. Furthermore, the literature shows that different tools are better at different verification tasks. In particular, targeted security-oriented tools are better at analyzing cryptographic components (shown in green in Figure 1) while standard software verification tools are effective on the rest. Combining these tools to verify a full protocol implementation remains a challenge.

hax: Verifying Rust Code with Multiple Provers. In this paper, we target protocol implementations written in idiomatic Rust, and we aim to drive proofs for all the protocol components from a single framework, while still using the best tool for each task. To this end, we use and build upon hax [14], a generic formal verification framework for Rust programs that translates the source code into the input languages of multiple backend provers, including F⋆, Rocq, ProVerif, and SSProve.

The programmer controls which tools are used to verify each module, and provides annotations in the Rust code that serve as proof goals and hints. Hence, the same code can be verified for different properties using different tools. hax supports safe Rust, which already guarantees memory-safety and type safety. This is a great improvement over C code. However, Rust code can still raise runtime exceptions ('panic'), e.g. when by an integer overflow or index out of bounds access of a vector. For `Bert13`, we use the F⋆ backend to prove runtime safety (the program does not crash/-panic) and to prove the correctness of message formatting; we use ProVerif to analyze the symbolic security of the core protocol logic and state machine code; we use SSProve to prove the computational security of protocol-specific cryptographic constructions.

Case Study: Formally Verifying `Bert13`. We demonstrate this methodology on `Bert13`, an implementation of TLS 1.3 that is written in Rust and supports both classical and post-quantum ciphersuites.[4]

`Bert13` uses formally verified cryptography from the libcrux library [41] and is practical on low-end devices with sub-10ms handshake completion, depending on the choice of ciphersuite. The core protocol code in `Bert13` is formally verified for the expected authenticity and confidentiality guarantees of TLS using ProVerif. The ProVerif model assumes the security of the key schedule, which we separately prove using SSProve. The model also assumes the correctness of the parsing code, which we verify using F⋆. Finally, we prove that the implementation does not panic at runtime, by verifying it for runtime safety using F⋆. In addition, we use the strong typing of the Rust type system to enforce coding disciplines such as secret independence and state machine linearity.

Contributions. In combination, these results are the first of their kind for cryptographic protocol implementations written in Rust, and `Bert13` is the first high-assurance implementation for a post-quantum variant of TLS. Ours is also the first machine-checked proof of the TLS 1.3 key schedule. We believe that the wide range of techniques we demonstrate in this paper will be independently useful as a guide to the formal analysis of other protocol libraries.

Outline. Section 2 outlines our multi-prover methodology for verifying Rust code. Section 3 describes the TLS 1.3 protocol and sets out verification goals for its implemen-

[4] We use an anonymous name for `Bert13`, which is developed as an open-source project, and will be de-anonymized before publication.

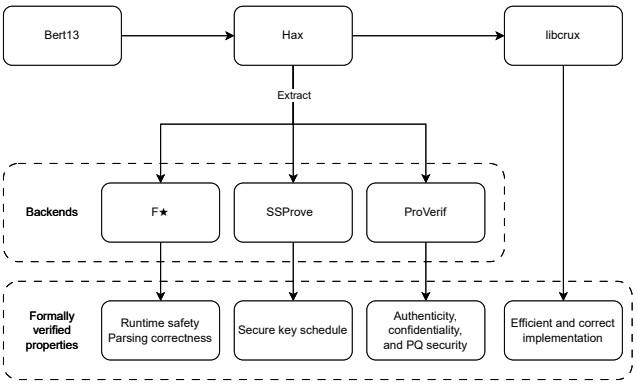


Figure 2: Verifying Protocol Implementations with hax

tations. Section 4 describes the `Bert13` implementation of post-quantum TLS 1.3. Section 5 proves security of the key schedule in the computational model with SSProve. Section 6 proves the main confidentiality and authentication guarantees for the `Bert13` code using the symbolic prover ProVerif. Section 7 uses F⋆ to prove runtime safety and message formatting properties for `Bert13`. Section 7.2 concludes with some discussion.

2 Methodology: Verifying Rust Code with hax

Our methodology is based on hax [14], a framework for Rust verification that supports multiple proof backends. The way we use hax in this paper is depicted in Figure 2. We begin with a Rust implementation of some cryptographic protocol (here `Bert13`). The implementation is written in idiomatic Rust but is annotated by the Rust developer with verification goals and proof hints. The hax toolchain takes the Rust code along with the annotations and translates it into the input language of different provers, where they can be verified for security or functional properties. Notably, the developer can choose which source modules are analyzed with which tools and for which properties. The cryptography underlying the protocol implementation is provided by libcrux, a formally verified cryptographic library.

The hax toolchain has been used before for security proofs of cryptographic constructions [37] and for the correctness proofs in libcrux itself, but this is the first work to apply hax to protocol implementations. The main advantage of hax for our work is that it allows us to use multiple provers while allowing the Rust developer to drive the verification. There are many other Rust verification tools under active development [39, 6, 27, 46, 59, 45, 34, 31, 64]. We chose hax for this project primarily for its support of both security and functional verification tools. On the other hand, hax itself does not support all of Rust, and so the developer has to stay within the supported subset to use the toolchain.

For each input Rust crate, hax parses it using the rustc compiler, performs a series of transformations to facilitate translation to the functional languages in proof assistants like F⋆ and Rocq, and then generates models for various backends.

F⋆ Backend. The first backend we consider is the F⋆ [57] proof assistant, which has been used in a number of verification projects including the HACL⋆ cryptographic library [62] and libcrux; see also Section 4.6. Verification in F⋆ proceeds with the aid of assertions, refinement types and invariants. This could be used to show that e.g. that the reverse function on lists preserves its length, or that QuickSort is indeed a correct sorting function. Such properties are proven with the help of the Z3 SMT-solver.

The Rust developer can write contracts in the form of pre- and post-conditions, assertions, and loop invariants, that are translated by hax into the appropriate verification conditions in F⋆. In particular, we typically use pre-conditions to provide constraints for runtime safety, and we use post-conditions to specify correctness properties. Once all functions are annotated with contracts, they can be verified by F⋆, for the most part automatically, using its SMT solvers, although some proofs may require some additional hints. (See Section 7 for how this works in `Bert13`).

This is in line with the very recent (*experimental*) addition of Contracts to the Rust language.[5] It envisions a unified language for static and dynamic checks, with the ultimate goal that:

> All unsafe functions in Rust should have their safety conditions specified using contracts, and verified that those conditions are enough to guarantee absence of undefined behavior. We provide an example in Section 4.4.
>
> Rust users should be able to check that their code do not violate the safety contracts of unsafe functions, which would rule out the possibility that their applications could have a safety bug.

In this work we show that hax already facilitates this for a realistic project such as `Bert13`. We recommend that the hax team aligns their contracts with the experimental contracts supported by the rust language, once their design stabilizes.

Rocq and SSProve backends. Rocq [58], like F⋆, is a general proof assistant build on dependent type theory. It is a foundational proof assistant in the LCF tradition. It does not use SMT-solvers, so we expect more user interaction would be required to prove runtime safety.

Our main use of Rocq is via the SSProve [38] library in Rocq which includes syntax, semantics and programming logic of a probabilistic imperative programming language, as commonly used by cryptographers in the computational model. It also provides a program logic in the spirit of EasyCrypt [10]. On top of this, it builds an interpretation of the State-Separating Proof [23] modular style of reasoning, also used in the Joy of Cryptography book [55]. The main ingredient of SSP is a calculus for program fragments (packages) in the aforementioned programming language.

[5] `https://rust-lang.github.io/rust-project-goals/2025h1/std-contracts.html`

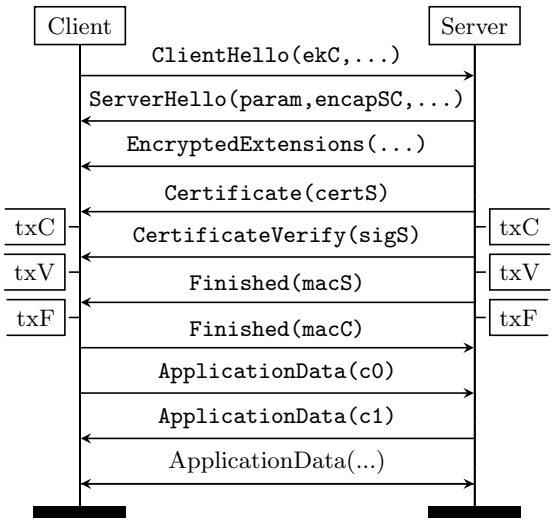
hax supports special annotations for cryptographic properties used in SSProve. These properties can then be proven in dialogue with a proof engineer. We will see in Section 5.4 that the SSP structure can help to improve the structure of the rust code.

By limiting the number of transformations in the hax toolchain, one can translate the hax subset of Rust to the simple imperative language used by SSProve in Rocq. hax even provides a proof that the functional and imperative translation agree. This can be seen as a partial correctness proof of the hax transformations.

ProVerif backend. In addition to proof assistants, hax also supports security verification of Rust code using dedicated protocol verifiers. Currently, it supports the ProVerif [21] tool, but others can be added similarly.

ProVerif is an automated tool for checking security protocols in the symbolic model (or 'Dolev-Yao'), which is codified using the applied π-calculus. Given security properties, such as confidentiality, integrity and authenticity, ProVerif will try to automatically verify these properties for a protocol model written in terms of message-passing processes. The symbolic model is less precise than the computational model (used in SSProve). It treats cryptographic primitives, such as encryption, as perfect black boxes. However, this has the advantage of much better automation. The two models aid each other, in the sense that one can prove in the computational model that the primitives have the assumed security properties.

Symbolic analyses, using tools like ProVerif, Tamarin, and DY*, have proved effective for the comprehensive formal analysis of real-world protocols like TLS, Messaging Layer Security, Noise, and Signal [42, 24, 60, 40]. We use ProVerif (in Section 6) to formally analyze our Rust implementation of TLS 1.3.



The main cryptographic computations in the protocol are:

```
(k,encapSC) =  KEM-Encap(ekC)
sigS =         Sign(skS, txC)
macS =         MAC(mkS, txV)
macC =         MAC(mkC, txF)
c0 =           AEAD(akC, m0)
c1 =           AEAD(akS, m1)
```

and the symmetric keys `mkC`, `mkS`, `akC`, `akS` are derived from the encapsulated key `k` via the key-schedule, as described in Section 5.

Figure 3: The TLS 1.3 protocol: main elements of the server-authenticated handshake and application data exchange

3 The TLS 1.3 Protocol

The Transport Layer Security (TLS) protocol is the IETF standard that that underlies all secure Web connections. In 2018, partly in response to some weaknesses in the previous protocol version, it was completely redesigned as TLS 1.3 [54].

3.1 Protocol Flow

Figure 3 shows the main protocol flow commonly used on the Web. The protocol is initiated by a *client* when it wishes to establish a connection with a *server*. The protocol starts with a key exchange, called the *handshake*, which authenticates the server (and potentially the client) and establishes a sequence of keys shared between them, via a novel cryptographic construction called the *key schedule*. Once the handshake is complete, the client and server can use the established keys to exchange encrypted application data with each other, using the *record* sub-protocol.

In Figure 3, the server is authenticated with an X.509 certificate, but the client remains unauthenticated. There is an alternate flow, not shown here, where the client also provides an X.509 certificate, and yet another without certificates, where both client and server authenticate each other via a pre-shared key. We purposely choose a KEM-based presentation of the protocol to make it possible to uniformly account for both Diffie-Hellman and Post-Quantum KEM-based key exchange modes.

The client first sends a `ClientHello` message containing an ephemeral KEM public key `ekC`. In response, the server sends a `ServerHello` containing a fresh key `k` encapsulated under `ekC`. After the `ServerHello`, both the client and server initialize the key schedule with the key `k`, and then use it to derive a sequence of keys as the protocol proceeds. For example, the key schedule produces handshake encryption keys, which are used to protect all subsequent handshake messages (a detail elided in the figure.)

The two hello messages also implement the negotiation phase of the protocol: the client offers a choice of versions,

ciphersuites, and other extensions, and the server chooses one set of parameters in its response and in the subsequent `EncryptedExtensions` message.

The server then sends its X.509 certificate in a `Certificate` message and proves that it knows the corresponding private key by providing a signature over the current protocol transcript in a subsequent `CertificateVerify` message. The server then ends its side of the handshake by sending a `Finished` message containing a MAC over the current transcript using a MAC key `mkS` derived from the key schedule.

The client processes this stream of handshake messages from the server, decapsulates the key `k` and derives the same sequence of keys from the key schedule to decrypt the handshake messages. It then validates the server's X.509 certificate using its local certificate validation library, and verifies the server's signature and MAC. It then sends its own `Finished` message to complete the handshake.

At this point, the client and server can start exchanging application data messages that are encrypted using AEAD keys for the two directions derived from the key schedule.

3.2 Formal Analyses of TLS 1.3

Given its importance to the Web ecosystem, TLS has been comprehensively analyzed against a variety of threats in a number of security models. For TLS 1.3, there are many pen-and-paper proofs of security (see e.g. [29, 22]), mostly focused on the core protocol logic and crypto constructions. There are also several machine-checked proofs of the protocol: proofs using symbolic provers like ProVerif [42] and Tamarin [24] that treat the cryptographic primitives abstractly using equational theories, and proofs using computational provers like CryptoVerif [42] and Computational F⋆ [26] that precisely model cryptographic algorithms as probabilistic functions over bit-strings.

All the proofs above are for abstract *models* of the protocol; they do not consider the precise cryptographic formats specified in the standard, or account for multiple ciphersuites running in parallel. Consequently, it is possible that they miss some attacks. Conversely, modeling and analyzing a large protocol like TLS 1.3 is not an easy task, and the risk that the model itself will have mistakes is non-trivial.

Consequently, we advocate that protocol security analysis must be performed, where possible, directly on the protocol implementation. In this way, one can be sure of not missing some low-level formatting detail, or some protocol feature that is needed for the normal functioning of the protocol. In the past, some works have analyzed reference implementations of TLS 1.3, such as a proof-of-concept JavaScript implementation [42] of the full protocol, and a verification-oriented F⋆ implementation [26] of the record layer. Neither of these are practical implementations; they were written primarily by researchers to exercise verification tools.

3.3 Goals for our TLS 1.3 Implementation

In this paper, our goal is to verify a practical Rust implementation of TLS 1.3. Our implementation must run efficiently on a variety of platforms, ranging from IoT devices, phones, desktops, to servers. It must interoperate with other TLS implementations including popular web browsers and web servers. Furthermore, it should support both classical Elliptic-Curve Diffie-Hellman key exchanges and post-quantum key exchanges (based on post-quantum KEMs).

Importantly, we would like to formally verify that the protocol implementation achieves the confidentiality and authentication guarantees expected of TLS. To achieve this proof, we need to make some assumptions about the underlying cryptography. TLS 1.3 mainly uses well-understood cryptographic constructions (signatures, MACs, AEAD encryptions) for which we can make standard assumptions. The only novel construction in the protocol is its *key schedule*, which needs new analysis. Furthermore, most of the cryptographic operations in the protocol rely on using the protocol *transcript* to encode all the session content, and so we must prove that the transcript is *unambiguous*: if a client and server have the same transcript, their view of the session (parameters, certificates, public keys, etc.) should be the same.

We summarize these classic protocol security requirements for TLS 1.3 implementations as follows:

- **Protocol Security Guarantees**: the protocol implementation must ensure that the server (and optionally client) is authenticated and that the application data sent between honest clients and servers is confidential. This, in turn, relies on two sub-goals.
  - **Key Schedule Security**: the cryptographic construction implemented in the key schedule implementation must be provably secure.
  - **Unambiguous Transcripts**: the transcripts maintained in the protocol implementation must be injective with respect to session data.

Beyond these core cryptographic security guarantees, a cryptographic protocol implementation must satisfy certain other functional properties that are also important for the security of the user. The implementation must be memory safe, i.e. it must not read or write data out of bounds, which might leak secrets (e.g. see HeartBleed[6]). It must not crash with an unexpected error, even if an adversary were to send a maliciously crafted message, otherwise it may enable denial-of-service attacks. It must implement the protocol state machine correctly and not accept or reject messages out of turn, or else it might open up state machine attacks [12]. And it must safely handle the ephemeral session secrets generated during the run of the protocol and not accidentally reveal them to the adversary.

We summarize these additional requirements for TLS 1.3 implementations as follows:

[6] `https://heartbleed.com`

- **Implementation Security Guarantees**: the implementation must not break the security invariants expected by the protocol application. In particular:
  - **Runtime Safety**: the protocol implementation must be memory safe and must not crash with an unexpected error.
  - **Session Secret Management**: the short-term secrets generated during a session must not be revealed to the attacker via some public channel.
  - **State Machine Correctness**: the implementation must correctly implement the protocol state machine

Of course, this list of properties is not complete. One may, for example, also wish to prove full functional conformance for the protocol implementation against a formal specification of the protocol. Here, we restrict our ambitions to proving properties we deem to be essential for security, based on known attacks on TLS implementations, and leave other properties for future work.

3.4 Implementation and Proofs

In Section 4, we present `Bert13`, our portable post-quantum TLS 1.3 implementation in Rust. Via interoperability testing, we experimentally verify that this implementation conforms to the TLS standard. In the implementation, we use the strong type system of Rust to enforce disciplines such as secret independence (for session secret management) and for state machine correctness.

In Section 5, we prove cryptographic Security for the key schedule implementation in `Bert13` using the SSProve tool. In Section 6, we prove the main confidentiality and authentication guarantees for the protocol code in `Bert13` using the symbolic prover ProVerif. In Section 7, we first use the F⋆ framework to prove the runtime safety for the entire protocol implementation. We then use F⋆ to also prove the transcript unambiguity for our implementation.

4 `Bert13`: Post-Quantum TLS 1.3 in Rust

`Bert13` is an implementation of the TLS 1.3 protocol intended for real-world usage. It is not intended to be a research artifact. As such, we have different requirements and approach development and verification as equally important goals. Hence, instead of writing the protocol in a proof-oriented language, which requires verification experts, `Bert13` is written in Rust, by Rust engineers. This illustrates our methodology of enabling domain experts and software engineers to work together towards a verified implementation.

4.1 Code Structure

The `Bert13` source code is separated into the core TLS 1.3 protocol and the necessary networking APIs. The repository also defines example client and server applications and provides utilities for interoperability testing.

The main components of the protocol implementation are as follows: the `formats` module implements the parsing and generation of TLS 1.3 messages; the `keyschedule` module and its submodules implement the key schedule; the `handshake` module implements the TLS 1.3 state machine and the main messaging functions for the handshake protocol; the `record` module implements the record layer encryption and decryption functions; and the `api` module provides a protocol API to applications.

The implementation relies on a few external libraries. The cryptography module provides a wrapper around the libcrux library, which provides verified implementations of all the necessary cryptographic primitives. One difference to classical TLS implementations is that the crypto module provides a Key encapsulation mechanism (KEM) API instead of Elliptic Curve Diffie Hellman (ECDH), to facilitate a uniform interface which captures both classical and Post-Quantum cipher suites.

Finally, the certificate module implements the minimal functionality required for parsing certificates as part of the TLS 1.3 handshake, and is considered an untrusted module here. The client application is expected to take the certificate, server name, and public key provided by the protocol API and validate them using an external PKI implementation. On the server, application needs to provide the protocol implementation with the appropriate certificate and private key.

4.2 Rust Types for Secret Independence

The protocol implementation uses the strong typing discipline of Rust to enforce several security and functional invariants.

Although `Bert13` relies on libcrux for all its cryptography, it must still carefully handle several secret values, such as the certificate private key (on the server) and various symmetric keys derived by the key schedule. To ensure that we do not inadvertently leak these values to the adversary, we use the Rust type system to enforce *secret independence*. When the feature `secret-integers` is set, all the byte-strings in `Bert13` are treated as potentially secret values. This means that their contents cannot be inspected, compared, written on public channels, or used as indices into arrays. Everything handled by the protocol is secret by default, and if the programmer wishes to look into a byte-string (because they know its contents are public) they must call the `declassify` function.

For example, after record encryption a ciphertext needs to be declassified before it can be sent on the network, and we can decide that this is safe because of the protocol security guarantees. Conversely, when decrypting a record, if we wish to inspect any part of the message, we must first declassify it, hence declaring that we are consciously willing to leak these contents. We enforce this strict discipline throughout the `Bert13` implementation.

Table 1: `Bert13` performance measurements across 1000 iterations.

Ciphersuite			Client Handshake Performance	
Cipher	Signatures	KEM	Time/Handshake [μs]	Throughput [per second]
Chacha20Poly1305	P-256 ECDSA	P-256 ECDH	8872	112.70
Chacha20Poly1305	P-256 ECDSA	X25519	5287	189.11
Chacha20Poly1305	P-256 ECDSA	X25519Kyber768Draft00	6275	159.34
Chacha20Poly1305	P-256 ECDSA	X25519MlKem768	6185	161.67

4.3 Rust Types for State Machines

We also rely on the linearity guarantees of Rust types to implement the TLS 1.3 handshake state machine. When each message is sent or received, the client or server retrieves its previous state and generates a new state. By using the Rust type system, we can enforce that the previous state has been *consumed* and hence cannot be used again. For example, the `put_server_hello` function which processes a server hello message has the following structure:

```
fn put_server_hello(
    handshake: &HandshakeData,
    state: ClientPostClientHello,
    ks: &mut TLSkeyscheduler,
) -> Result<(DuplexCipherStateH,
  ClientPostServerHello), TLSError> {
    let ClientPostClientHello(...) = state;
    ...
    Ok((
      DuplexCipherStateH::new(...),
      ClientPostServerHello(...)))
}
```

In Rust, the argument `state` is not a pointer, the caller is transferring ownership of the state to this function which is consuming the old state and creating a new one. The caller cannot use the old `state` after calling this function. This style of implementing state machines is sometimes called *type state* and is usable in any language that provides *affine types* like Rust does. We implement the entire handshake state machine in this style.

4.4 Developer-driven Proof Annotations

The software engineers writing the Rust code can also add pre-conditions to help with the verification. In some areas this enforces some safe engineering practices that are otherwise only enforced by reviews. Take for example the length check in the listing below. The default way of comparing the lengths would panic, which will most likely not be caught in tests. Fuzzing may catch bugs like this. But the verification statically *ensures* that this check does not over- or underflow. The software engineer can make sure of this by adding the "requires" before the function and use the correct way of comparing the length.

```
#[requires(self.len() >= start)]
pub(crate) fn find_handshake_message(
  &self,
  handshake_type: HandshakeType,
  start: usize,
) -> bool {
  // self.len() < start + 4 would panic
  if self.len() - start < 4 {
      return false;
  }
```

4.5 Implementing Post-Quantum TLS 1.3

As mentioned before, `Bert13` supports both classical cipher suites and Post-Quantum cipher suites. Since `Bert13` uses a KEM based crypto API, supporting Post-Quantum cipher suites does not require changes to the protocol implementation.

`Bert13` implements the hybrid ciphersuite X25519MLKEM768 $0x11ec$[7] defined in [43]. Note that the exact hybrid specification for TLS 1.3 is still in progress. However, this ciphersuite is currently implemented by Firefox, Chrome, Cloudflare and others, and is compatible with the draft RFC Hybrid key exchange in TLS 1.3 [56]. The shared secret that is used to compute the TLS 1.3 master secret is defined as the 64 bytes concatenation of the X25519 shared secret and the ML-KEM shared secret shared_secret = X25519.shared_secret || ML-KEM.shared_secret.

4.6 libcrux: Formally Verified Cryptography

libcrux is a formally verified cryptographic library that provides all the primitives necessary for TLS 1.3 in `Bert13`. It contains code written in Rust and proven with hax [14], as well as verified Rust code generated from the HACL⋆ project [63, 33]. It provides, in particular, its own verified Rust implementation of ML-KEM, that is used to provided support for hybrid post-quantum KEMs in `Bert13`.

Each algorithm implemented in libcrux is formally verified for runtime safety (memory safety and crash freedom), for functional correctness with respect to a high-level mathematical specification of the algorithm written in F⋆, and

[7] `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml`

for secret independence, a discipline that prevents certain classes of side channels. Despite including only verified implementations, code from libcrux is often as fast as or faster than unverified cryptographic implementations.

4.7 Performance and Interoperability

`Bert13` is portable across all `std` targets supported by the Rust compiler, and the libcrux library. Bare metal `no_std` environments are supported in the presence of a global allocator. The implementation is compatible with Chrome (134), Firefox (137), and Cloudflare on all implemented ciphersuites.

The `Bert13` library supports the following algorithms and protocols

as signature schemes: RSA-PSS-SHA256, ECDSA-P256-SHA256, Ed25519
as KEM: X25519, P-256 ECDH, X25519Kyber768-Draft00, X25519MlKem768
as session cipher: Chacha20Poly1305
as digest: SHA256, SHA384, SHA512

Other ciphersuites such as AES-GCM can be supported when using for example HACL⋆-backed C bindings instead of the pure Rust implementations used here. While there may be faster cryptographic implementations out there, the performance numbers in Table 1 show that `Bert13` is a usable implementation with performance comparable to the most popular TLS libraries.

See Table 1 for `Bert13` client benchmark results obtained on a Raspberry Pi 3 Model B Rev 1.2, with 900 MB of RAM and a Broadcom BCM2835 CPU running at 1.2 GHz. On this device, to establish a connection, the client 55.8 KB of stack memory using X25519 as KEM and 85.1 KB using a post-quantum hybrid KEM, at a binary size of 2980 KB.

5 Key Schedule Security with SSProve

One of the essential parts of the TLS protocol is the key scheduler. It is responsible for generating secure keys used throughout the communication between the client and server, and for eliminating incorrect or invalid invocation of key generation. An example of an attack on the key schedule is tricking the key schedule to generate the same key for two different parts of the protocol. Another type of attack is not including enough randomness or new information into the key generation. Thus making the newly generated keys weaker than required. To mitigate these attacks, we ensure that the implementation in `Bert13` is covered by the security proof from [22].

5.1 State-Separating Proofs (SSP)

The core theorem in [22] is a security proof bounding the advantage of an adversary to distinguish between a key generated by invoking the key schedule and a uniformly random key.

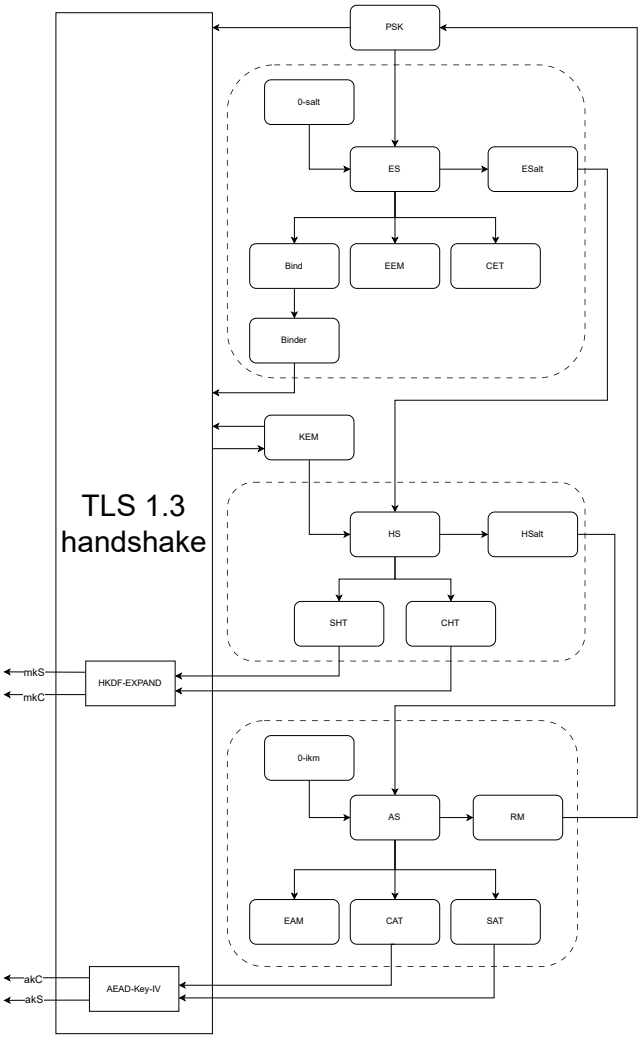


Figure 4: Calls to key schedule in the handshake protocol

The paper also proves two other theorems. The modular theorem states that one can introduce a mapping of keys. This allows a more abstract treatment, thus simplifying the arguments in the core theorem. The main theorem states the security of the composition of the modular games is bounded by a more classical monolithic version of the game, thus ensuring we can reason about the parts and still get a security statement for the protocol as a whole.

All the theorems have a pen-and-paper proof [22] in the state-separating proof (SSP) style [23]. The benefits of using SSP are that it enables modular reasoning, which is helpful when trying to scale to a large protocol like key schedule for TLS 1.3. This is achieved by providing a clear interface for each module (or 'package'). These modules can be composed in serial or parallel to create larger and more advanced packages. Security is shown by using security games: Given two packages, without any imports, one shows that an adversary cannot distinguish between them (up to negligible probability). One package describes the real behavior of the protocol and the other describes the ideal behavior. A 'game hop' replaces the real package with the ideal package. The entire protocol is defined as the composition of such packages. By a sequence of game hops, one idealizes the protocol step by step.

5.2 Mechanizing SSP in SSProve

In this paper we focus on formalizing the core theorem in SSProve. SSProve is a foundational framework in Rocq for modular cryptographic proofs in the SSP style [1].

We write the key schedule in Rust and translate the code into SSProve using hax. This guarantees that not only that the abstract Key schedule protocol is secure, but also its Rust *implementation*. We prove this by showing functional equivalence between the implementation and the package specifying the real behavior of the protocol. The equivalence is another game. We obtain the security guarantees of the implementation by transitivity.

5.3 The Formalization

The overall structure for the proof of the core theorem is given by two hybrid arguments. These come naturally from the package composition structure. The key schedule protocol is defined in a number of rounds. One of the hybrid arguments shows that one can idealize one round at a time. More concretely, we define a package for a single round of the key schedule parameterized by the round number. The key schedule package is then given by the composition of the rounds in serial, since we have a dependence on the keys of the previous round. The second hybridization argument comes from the structure of the round itself. The idealization order from [22] ensures that we can split the round into groups of packages. Each group has no dependence internally and only depends on packages earlier in the idealization order. This closely mirrors the steps in the handshake protocol, as no extra communication is needed to generate all keys in a group. We only need additional information when generating the next group in the order. The hybridization argument states that: from a bound on the advantage of idealizing each type of package, we obtain a bound on idealizing the entire round.

The proof [22] first uses the hybrid argument for proving a bound on the rounds (horizontal) and then the hybrid argument for the full protocol as a bound on the round number (vertical). However, during the formalization, we realized that we can swap the order of the hybrid arguments — do the vertical proof first for each of the smaller key packages, and then do the horizontal proof. One reason to do this is that the vertical proofs are simpler, though more plentiful. In the last round of the protocol, we do not generate the preshare key (PSK) for the next round, so there is some difference in the interface description for the horizontal package. By swapping the order, we can handle this misalignment directly, as the horizontal proof only needs to align with the package interface of the full protocol when it is the outer hybrid argument.

5.4 The Implementation

The implementation of the key schedule is written in Rust. To facilitate the equivalence proof, we modified the implementation of the key schedule to follow the modular structure in the proof. That is, we wrote functions and interfaces based on the description in the state-separating proof (SSP) packages. This facilitates equivalence proofs, as we just have to bundle the functions into packages and then show equivalence to the SSProve package line-by-line. Moreover, it clarifies and modularizes the code base. This use of SSP for structuring implementations is one of our contributions. The rewrite of the key schedule made the handshake protocol improved readability and highlighted some shortcomings of the initial `Bert13` implementation.

Echoing the Curry-Howard correspondence, (cryptographic) proofs are programs, thus they need to be modular and parametric. Moreover, to maintain verified code, we should ensure that the code is close to the specification used by the proofs; thus, the structure of the proof guides the structure of the code and visa versa. Working with SSP is beneficial to this process, as SSP ensures modularity of proofs and code in the specification, which can be mirrored by the implementation.

The proof suggest implementing the four functions: `PrntN`, which encodes the transition graph as a map to the parent keys needed to produce a given key; `label` maps a key to its label and is used in `XTR` and `XPD` to ensure correctness of the key state; `XTR` runs key extraction (e.g. `HKDF-EXTRACT`), used when there are two parent keys; `XPD` runs key expansion (e.g. `HKDF-EXPAND`), used when there is only one parent key.

These functions together complete the graph in Fig. 4, thus implementing the key schedule for TLS. Some computations can be bundled together, so we compute/derive their values in rounds. This more or less follows the groupings

generated by the idealization order. For `XTR` we combine two keys and a label, while `XPD` takes one key and some data.

5.5 Formalization Effort

The following gives a crude overview of the formalization/implementation effort.

- The paper proof in [22] (∼ 1600 LoT)
- The security proof (∼ 7500 LoC)
- The Rust implementation (∼ 700 LoC)
- The translation (∼ 1200 LoC)

The formalization is a little more than 4 times the length of the informal proof, which is reasonable, given that the formalization is more detailed.

We conjure that it is possible to facilitate the proof process by automation. The composition proofs are especially well suited for automation, as most of the proofs are boilerplate based on the structure of the composition. We also spend some effort to argue about disjointedness of package. This could possibly be simplified using Nominal SSProve [44] saving about 500 LoC.

The translation of the code is quite close to the original code, so the size difference is quite small, which is one of the benefits of using hax over other tools.

5.6 Security Reduction

The security proof in SSProve follows the pen-and-paper proof [22], which uses Diffie-Hellman for key exchange. Instead `Bert13` uses a KEM based version of TLS, which is suitable for agile cryptography as it generalizes both DH and ML-KEM.

We prove security of `Bert13` assuming an IND-CCA secure KEM, such a KEM is provided by a DHKEM or ML-KEM [4]. The TLS key schedule paper [22, Sec. 6], already suggest this is possible. For both DHKEM and ML-KEM libcrux provides verified rust implementations. We assume that the ML-KEM implementation in libcrux[8] agrees with the ML-KEM specification in EasyCrypt. The latter has been verified to be cryptographically secure [4].

We proved the Core Key Schedule Theorem [22, Appendix. D], which guarantees that the generated keys remain private. This theorem follows from six lemmas, D2-7 [22, Fig. 17]. We prove the main lemma D6. The others are direct consequence of the correct implementation of the cryptographic primitives which we inherit from libcrux.

To sum up, we have reduced the security of `Bert13` to the existence of a secure hash function, such as provided by libcrux. We also rely on libcrux for secure cryptographic primitives such as HKDF-EXTRACT and HKDF-EXPAND.

[8] `https://cryspen.com/post/ml-kem-implementation/`

6 Verifying the Protocol Code with ProVerif

ProVerif [21] is an automated tool for protocol verification in the symbolic model, also known as the Dolev-Yao model [28, 49]. In conventional use of the tool, designers model by hand their protocol in a process calculus, where cryptographic primitives are treated in an idealized fashion as constructors and destructors on terms.

ProVerif then allows protocol designers to formulate queries on trace properties that should hold on all protocol runs, e.g. the occurrence of a certain event in the trace should imply previous occurrence in the trace of another event, or certain events should be ruled out for all traces. This allows, among others, a natural formulation of authentication and confidentiality guarantees as properties off the set of possible protocol traces.

Namely, if the trace contains an event indicating that a server has concluded a handshake with a client, obtaining a session key in the process, we can ask ProVerif to verify that in all traces this event is preceded by another event indicating that the client has initiated a handshake with the server and that in no trace will the session key be revealed to the attacker. Such properties can be strengthened by adding expected failure modes, e.g. explicitly allowing the attacker to learn the server's longterm secret keys.

We use the hax toolchain to automatically extract a ProVerif model of the TLS 1.3 handshake from `Bert13`. We then write, by hand, the top-level processes that define the protocol scenario and the security queries that encode the verification goals.

6.1 Generated Protocol Model

For each protocol function in the source Rust code, hax generates a ProVerif function modeling its behavior. For example, the Rust function `put_server_hello` is used by the client to process the server's hello message. It gets translated to a ProVerif function:

```
letfun put_server_hello(
   msg : t_HandshakeData,
   state : t_ClientPostClientHello,
   ks : t_TLSkeyscheduler)
=
   let ClientPostClientHello(
       client_random, ciphersuite, sk, psk, tx)
       = state in
   let (sr: t_Bytes, ct: t_Bytes) =
       parse_server_hello(ciphersuite, msg) in
   let shared_secret =
       kem_decap(ciphersuite, ct, sk) in
   let tx = transcript_add(tx, msg) in
   let th = transcript_hash(tx) in
   let shared_secret_handle = key_schedule(...)
```

This handshake function takes three arguments, an input handshake message `msg`, an input `state`, and a handle to the key schedule `ks`. It first opens up the input state (which must be the state after sending the client hello) to extract the current session parameters; it then calls the `parse_server_hello` function to parse the incoming message as a server hello. If parsing succeeds, it computes the `shared_secret` by calling `kem_decap`, updates the transcript hash and starts deriving keys with the key schedule.

The main thing to note here is that the ProVerif model captures the flow of the Rust code, including the state management, cryptographic calls, and calls to the message formatting and key schedule functions. We model exactly what the Rust code does, and do not miss any branch or coding detail.

In total, we translate 104 Rust types to ProVerif types and 119 functions from Rust to ProVerif constructors, destructors or process macros, resulting in a generated model of 5980 lines. This mainly covers the handshake and record protocols.

However, the underlying libraries are abstracted in our ProVerif model: the cryptographic library models KEM encapsulation and decapsulation using symbolic constructors and destructors; the messaging formatting model treats serialization functions as constructors and parsing functions as destructors, without modeling the precise bit-level formats of these messages; and the key schedule model uses expand and extract as opaque constructors. These abstractions are standard for symbolic analysis, but in this paper, we justify these assumptions wherever possible, by developing proofs in SSProve and F⋆, and by relying on the correctness of the underlying libcrux crypto library.

6.2 Hand-written Verification Scenario

To complete our protocol model, we write by hand a top-level process that composes several sub-processes:

- `CreateServer` sets up server long term secrets corresponding to the ciphersuite given as an argument.
- `Client` models a client that connects to a server using a specified ciphersuite; it models the client handshake state-machine by calling the generated protocol functions (like `put_server_hello`) in sequence.
- `Server` which accepts connections from clients; it reads long-terms secrets from a table populated by `CreateServer` and then calls a sequence of server-side functions generated from the Rust code.
- `CompromiseServer` which allows the attacker to compromise server long-term secrets based on the server name, thereby emitting a `LeakServerCertSK` event in the trace.

```
process
      !CreateServer(SHA256_Chacha20Poly1305...)
      (* ... *)
    | !Client(SHA256_Chacha20Poly1305...)
      (* ... *)
    | !Server() | !CompromiseServerCertSK()
```

Each of these sub-processes is replicated, which means that we model an unbounded number of client and server sessions, and an unbounded number of compromises. We also allow clients and servers to run any non-PSK ciphersuite. The attacker is not specifically modeled; instead the ProVerif attacker is any process running in parallel to the protocol which can read and write on public channels and make use of its own keys as well as compromised keys, and can interfere with any number of sessions to try and break the security goals of the protocol. This sets up our verification scenario.

6.3 Protocol Analysis

Now that we have the protocol model, we can ask ProVerif to prove that the model provides *server authentication*, as well as *session key forward secrecy* for authenticated sessions.

At the end of the handshake, the client and server construct a duplex cipher state `cipher_state`, which contains among others the choice of AEAD algorithm, the client-to-server key `akC` as well as the server-to-client key pair `akS`. We write `cipher_state(akC)` to denote that `akC` is part of a cipher state. We state our security goals for the TLS handshake in terms of these cipherstates.

Server Authentication. We show that whenever a client finishes the handshake with a given server, the server must have finished as well, deriving the same cipherstate. This holds unless the server's long term certificate private key was compromised.

```
query
  server_name:  t_Bytes,
  cipher_state: t_DuplexCipherState,
  client_state: t_ClientPostClientFinished,
  server_state: t_ServerPostServerFinished;

  event(ClientFinished(server_name,
                       cipher_state,
                       client_state))
  ==>
   event(ServerFinished(server_name,
                        cipher_state,
                        server_state))
|| event(LeakServerCertSK(server_name)).
```

ProVerif shows verifies this query in under 2s.

If we ask ProVerif to prove that this query holds without the clause for server compromise, ProVerif finds an attack within 3 seconds that uses the compromised server key.

Session Key (Forward) Secrecy. We show that if the attacker learns a session key, then the server's long term certificate private key was compromised before the client was finished.

```
query
  i: time, j:time,
  server_name:  t_Bytes,
  cipher_state: t_DuplexCipherState,
  client_state: t_ClientPostClientFinished;

  event(ClientFinished(server_name,
                       cipher_state(akS),
                       client_state))@i
  && attacker(akS)
  ==>
   event(LeakServerCertSK(server_name))@j
   && i > j.
```

Hence, if the attacker learns the server's private key and uses it to impersonate the server, it may then learn the session key `akS` established in the session. In all other cases, the session keys are confidential. In particular, session keys established before the server compromise remain confidential.

Message integrity and confidentiality As corollaries of the handshake security goals above, we can also ask ProVerif to prove the integrity and confidentiality of each application data message sent or received in either direction.

6.4 Post-Quantum Security against Harvest-Now-Decrypt-Later Attacks

`Bert13` implements post-quantum ciphersuites for TLS 1.3 and so we also analyze whether the protocol is secure against a class of quantum adversaries. In particular, we model Harvest-Now-Decrypt-Later attackers, in the same way as prior work on symbolic analysis of post-quantum protocols [18].

We include in our model the possibility that at some time, marked by an event, the attacker is able to compromise all Diffie-Hellman constructions and signature algorithms. After this time, the attacker can obtain Diffie-Hellman private keys and forge signatures.

We then ask if our protocol model is still secure, if the KEM constrution is unaffected. ProVerif is able to prove that all the queries above still hold, as long as the quantum apocalypse occurs after the session is completed. In other words, as long as we use a PQ-KEM, a passive attacker today who records all messages cannot break the TLS 1.3 guarantees using a quantum computer in the future.

7 Verifying Runtime Safety and Unambiguous Message Formats with F⋆

Using the hax toolchain, we translate the full protocol implementation to purely functional code in F⋆. This includes all the key schedule code, the message formatting modules, the protocol state machine, and the core handshake and record protocol code, all the way up to the protocol API. The total amount of Rust code we process is 3264 lines (without comments) in 8 modules, which translate to 10964 lines of F⋆.

7.1 Runtime Safety

Rust is a memory-safe language equipped with a strong type system. The Rust borrow-checker enforces that mutable variables cannot be aliased, and is able to impose a strong discipline over the use of memory in a program. The hax toolchain relies on this discipline to translate Rust code with side-effects into purely functional F⋆.

However, although the Rust compiler ensures that safe Rust cannot access memory out of bounds, programs can still try, and this will result in a `panic`, an unrecoverable exception where the program essentially crashes. Other language features can also panic: for example, arithmetic over a machine integer that results in an out-of-bounds value is undefined behavior and will panic in debug builds, and so can calls to `unwrap` on a `Result` or `Option`.

When hax translates a potentially-panicking Rust function to F⋆, it requires the programmer to prove that the function is *total*, that is, it can never panic. For example, consider the funcion `find_handshake_message` excerpted in Section 4. When translated to F⋆, it has the following implementation:

```
let rec impl_HandshakeData__find_handshake_message
        (self: t_HandshakeData)
        (handshake_type: t_HandshakeType)
        (start: usize)
      =
      if ((impl_HandshakeData__len self <: usize) -!
            start <: usize) <. mk_usize 4
      then false
      else ...
```

Here, the function uses the strict subtraction operator `-!` which requires that the result of the subtraction must be within the bounds of the `usize` type, and hence cannot be negative. When we try to type-check this function in F⋆, F⋆ immediately flags an error saying that it found a situation when this subtraction might underflow.

However, when we add the relevant pre-condition to the Rust code, the generated F⋆ function has a type declaration as follows:

```
val impl_HandshakeData__find_handshake_message
        (self: t_HandshakeData)
        (handshake_type: t_HandshakeType)
        (start: usize)
      : Prims.Pure bool
        (requires (impl_HandshakeData__len self <: usize)
                  ≥ start)
```

With this pre-condition, F⋆ is able to automatically prove that the subtraction is safe and that the full function is panic-free.

In this case, the function was correct, we just needed a type annotation, but during the course of our verification we found a number of cases, usually in message parsing functions, where the code was allowing for panics and we needed to change it to ensure panic-freedom. This is particularly important for code that runs on inputs taken from the untrusted network, since the attacker may have send us a maliciously crafted message to crash our software to trigger a denial-of-service, or worse.

One example of such a function is the `parse_client_hello` function, the very first function a TLS server calls on data it receives over a connection. The body of the function looks as follows:

```
let version = bytes2(3, 3);
let mut next = 0;
check_eq_with_slice(version.as_raw(),
                    client_hello.as_raw(),
                    next, next + 2)?;
next += 2;
check(client_hello.len() >= next + 32)?;
let client_random =
    client_hello.slice_range(next..next + 32);
...
```

This code uses the variable `next` as a pointer into the input `client_hello`. It starts by calling `check_eq_with_slice` to check that the first two bytes of the input matches the expected protocol version (this function returns an error if the input is too short or the match fails). It then increments `next` by 2 and extracts the client random value by slicing the next 32 bytes of the `client_hello`, after checking that the input has a sufficient number of bytes.

In an earlier version of this function, there was no call to `check` before the client random was extracted. Consequently, an attacker could have sent any message of size less than 34 and crashed the server (with a panic). Verification with F⋆ finds this bug, and adding the check suffices to prevent it.

By adding a combination of such checks (when needed) and pre-conditions, we are able to prove that all 3K+ lines of the protocol implementation are panic free.

7.2 Proving Transcript Unambiguity

As discussed in Section 3.3, the security of the TLS handshake relies crucially on the protocol transcript unambigiously representing the contents of the handshake. However, in the ProVerif analysis of Section 6, we abstract away from the low-level formatting details of the handshake messages and transcript, instead simply treating them as symbolic constructors.

Abstracting away from message formats is quite usual in protocol security analyses; indeed, other machine-checked proofs of TLS 1.3 [42, 24] also make the same assumption, and so do all pen-and-paper proofs. The main reason for this assumption is that handling the bit-level formatting details is annoying and seems irrelevant to the cryptographic analysis of the protocol.

In this paper, we seek to verify protocol *implementations*, not abstract models, and so we need to justify this abstraction. Furthermore, as many recent works show, ambiguity in important cryptographic inputs, like the TLS 1.3 transcript, can sometimes lead to serious attacks and should not be ignored [61].

Consider the function that serializes the client hello:

```
#[cfg_attr(feature = "hax-pv", pv_constructor)]
pub(crate) fn client_hello(
    algorithms: &Algorithms,
    client_random: Random,
    kem_pk: &KemPk,
    server_name: &Bytes,
    session_ticket: &Option<Bytes>,
) -> Result<(HandshakeData, usize), TLSError> {
  ...
}
```

The annotation above the function says that this function is treated as a constructor in the ProVerif analysis; in other words, we assume that given a serialized client hello, we can unambiguously parse from it the `algorithms` the client offered to the server, the client random, the client's public key, the name of the server the client wished to connect to, and the session ticket pointing to the pre-shared key (if any).

This serialized client hello is added to the transcript at both client and server, and hence after authenticating the transcript in the `Finished` messages, we know that the client and server have the same view of these fields, which is crucial for a key agreement and negotiation protocol like the TLS handshake.

To justify the assumption that the `client_hello` function operates like an injective constructor, we add a second annotation to the function, this time a post-condition for use in the F⋆ backend:

```
#[hax_lib::ensures(|result| match result {
    Result::Ok((ch,trunc_len)) => {
      trunc_len <= ch.len() &&
      match parse_client_hello(algorithms, &ch) {
        Result::Ok((cr,_,sn,pk,st,_,_)) =>
        cr == client_random &&
        &pk == kem_pk &&
        &sn == server_name &&
        &st == session_ticket,
        _ => false }},
    _ => true})]
pub(crate) fn client_hello(...) {...}
```

Table 2: Formal Verification Results for `Bert13`

Backend Prover	Rust Modules	Rust LoC	Translated LoC	Properties Proven	Time Taken for Proofs (s)
SSProve	1	425	815	Core Key Schedule Security	11m17s
ProVerif	3	1723	5980	Forward Secrecy, Authentication HNDL Post-Quantum Security	20s
F⋆	8	3264	10964	Runtime Safety, Unambiguous Formats	1m21s

The `ensures` clause states that if the `client_hello` function succeeds and returns a serialized value `ch`, then if we parse this resulting value using the `parse_client_hello` function, we obtain the same values that were passed into `client_hello`. In other words, `parse_client_hello` works as an inverse of `client_hello`. So, if the client and server have the same transcript, and hence the same `client_hello`, they must also agree on all the inputs to the `client_hello` function. The post-condition also tracks other variables like `trunc_len` which we ignore here, but are needed for the panic-freedom proofs elsewhere in the protocol code.

This post-condition is then proved for the code of `client_hello` in F⋆. In a similar way, we annotate and prove unambiguity for all the message formats in TLS 1.3 and hence for the transcript.

8 Discussion

In this paper, we have demonstrated a verification methodology for cryptographic protocol implementations written in Rust. The key features of this methodology are that it targets code written by professional Rust developers (not verification researchers), and that we use multiple specialized provers to handle different parts of the proof, rather than rely on a single proof framework. In this way, we were able to prove both secrurity and functional properties for `Bert13`, our post-quantum TLS 1.3 implementation. Our implementation and all our proofs are provided in the submitted artifact.

The formal verification results for `Bert13` are summarized in Table 2. We used three tools: SSProve for cryptographic security of the key schedule code, ProVerif for the symbolic security of the protocol code, and F⋆ for runtime safety of the full protocol implementation and proofs about message formatting. Each tool is well-suited to its task, and this can be seen by the time and effort we spent on each task. Using a single framework for all proofs would have, we believe, suited one task but made the others much harder. Conversely, using a single framework has the advantage that the the properties proved for different parts of the coe can be formally connected with each other. We forego this benefit in favour of our pragmatic approach which makes it possible to effectively verify real-world Rust code.

Comparison with Other Approaches. We have already discussed a number of related works throughout the paper. Here, we focus on works that seek to verify cryptographic protocol implementations.

The miTLS project [17] developed a verified reference implementation of TLS 1.2 in a functional programming language, but this code was never considered a practical implementation.

Project Everest [13] was an umbrella project that sought to build a formally verified implementation of the entire HTTPS stack. The project produced verified cryptographic libraries [62, 52], message formatting libraries [53], and a TLS 1.3 implementation [26], all of which were written and verified in the F⋆ framework before being compiled to C. The generated C code was incorporated into many mainstream software projects and hence was used in production. However, the source code in F⋆ is arguably inscrutable to protocol developers, and the proofs for TLS 1.3 were incomplete, since they only covered the record layer, not the handshake.

RefTLS [42] used another compilation toolchain to compile a TLS 1.3 implementation written in JavaScript to models in ProVerif and CryptoVerif [20]. Hence, the authors were able to analyze the same protocol code in both the symbolic and computational models. However, the source code in JavaScript was not meant to be used in production, and the proofs did not include the message formatting code or guarantee runtime safety.

Implementations of protocols other than TLS have also been formally verified, including the Signal protocol [51], the Noise protocol framework [40], and messaging layer security [60]. All of these implementations are in functional languages, although some of them can be compiled to C or WebAssembly.

Future Work. We believe the methodology demonstrated in this paper is effective and flexible and can be extended with other verification tools and applied to other protocol implementations. In future work, we intend to explore the use of computational proof frameworks like EasyCrypt and CryptoVerif to verify stronger security properties for the protocol code than one can prove with ProVerif. We would also like to extend the functional verification guarantees beyond the proocol layer into the X.509 cerification library and the networking APIs. As the post-quantum transition gets into full swing, we believe formal verification techniques like the one presented in this paper will be essential for us to have confidence in the new set of protocols and their implementations.

References

[1] Carmine Abate, Philipp G. Haselwarter, Exequiel Rivas, Antoine Van Muylder, Théo Winterhalter, Catalin Hritcu, Kenji Maillard, and Bas Spitters. SSProve: A foundational framework for modular cryptographic proofs in Coq. In *CSF*, pages 1–15. IEEE, 2021.

[2] 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 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 5–17. ACM, 2015.

[3] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.

[4] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. Formally verifying Kyber. In *CRYPTO 2024*, pages 384–421. Springer, 2024.

[5] 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 *CCS*, pages 1807–1823. ACM, 2017.

[6] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 3, pages 147:1–147:30, 2019.

[7] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. DROWN: breaking TLS using sslv2. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 689–706. USENIX Association, 2016.

[8] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: malleable encryption goes awry. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 146–163. IEEE, 2023.

[9] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *SP*, pages 777–795. IEEE, 2021.

[10] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 146–166. Springer, 2013.

[11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8:1–8:45, 2011.

[12] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: taming the composite state machines of TLS. *Commun. ACM*, 60(2):99–107, 2017.

[13] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, K. Rustan M. Leino, Jay R. Lorch, Kenji Maillard, Jianyang Pan, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean Karim Zinzindohoue. Everest: Towards a verified, drop-in replacement of HTTPS. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[14] Karthikeyan Bhargavan, Maxime Buyse, Lucas Franceschino, Lasse Letager Hansen, Franziskus Kiefer, Jonas Schneider-Bensch, and Bas Spitters. hax: Verifying security-critical rust software using multiple provers. In *Verified Software. Theories, Tools and Experiments (VSTTE)*, 2024. `https://eprint.iacr.org/2025/142`.

[15] 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 *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 98–113. IEEE Computer Society, 2014.

[16] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Trans. Program. Lang. Syst.*, 31(1):5:1–5:61, 2008.

[17] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 445–459. IEEE Computer Society, 2013.

[18] Karthikeyan Bhargavan, Charlie Jacomme, Franziskus Kiefer, and Rolfe Schmidt. Formal verification of the PQXDH post-quantum key agreement protocol for end-to-end secure messaging. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[19] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript collision attacks: Breaking authentication in tls, IKE and SSH. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[20] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, volume 117, page 156, 2007.

[21] Bruno Blanchet. Automatic verification of security protocols in the symbolic model: The verifier proverif. In *FOSAD*, volume 8604 of *Lecture Notes in Computer Science*, pages 54–87. Springer, 2013.

[22] Chris Brzuska, Antoine Delignat-Lavaud, Christoph Egger, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. Key-schedule security for the TLS 1.3 standard. Cryptology ePrint Archive, Paper 2021/467, 2021.

[23] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *Advances in Cryptology – ASIACRYPT 2018*, page 222–249. Springer, 2018.

[24] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. 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 1773–1788. ACM, 2017.

[25] Joyanta Debnath, Christa Jenkins, Yuteng Sun, Sze Yiu Chau, and Omar Chowdhury. ARMOR: A formally verified implementation of X.509 certificate chain validation. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 1462–1480. IEEE, 2024.

[26] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Jonathan Protzenko, Aseem Rastogi, Nikhil Swamy, Santiago Zanella-Béguelin, Karthikeyan Bhargavan, Jianyang Pan, and Jean Karim Zinzindohoue. Implementing and proving the TLS 1.3 record layer. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 463–482. IEEE Computer Society, 2017.

[27] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. Creusot: a foundry for the deductive verification of rust programs. In *International Conference on Formal Engineering Methods*, pages 90–105. Springer, 2022.

[28] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Inf. Theory*, 29(2):198–207, 1983.

[29] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol. *J. Cryptol.*, 34(4):37, 2021.

[30] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020.

[31] Nima Rahimi Foroushaani and Bart Jacobs. Modular formal verification of rust programs with unsafe blocks, 2022.

[32] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 341–350. ACM, 2011.

[33] Aymeric Fromherz and Jonathan Protzenko. Compiling C to safe Rust, formalized, 2024.

[34] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. RefinedRust: A type system for high-assurance verification of rust programs. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1115–1139, 2024.

[35] Cesar Pereida García and Billy Bob Brumley. Constant-Time callees with Variable-Time callers. In

26th USENIX Security Symposium (USENIX Security 17), pages 83–98, Vancouver, BC, August 2017. USENIX Association.

[36] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. 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 38–49. ACM, 2012.

[37] Philipp G. Haselwarter, Benjamin Salling Hvass, Lasse Letager Hansen, Théo Winterhalter, Catalin Hritcu, and Bas Spitters. The last yard: Foundational end-to-end verification of high-speed cryptography. In *CPP*, pages 30–44. ACM, 2024.

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

[39] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *PACM PL*, 6(ICFP), 2022.

[40] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 107–124. IEEE, 2022.

[41] Franziskus Kiefer, Karthikeyan Bhargavan, Lucas Franceschino, Denis Merigoux, Lasse Letager Hansen, Bas Spitters, Manuel Barbosa, Antoine Séré, and Pierre-Yves Strub. HACSPEC: a gateway to high-assurance cryptography. *RealWorldCrypto*, 2023.

[42] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 435–450. IEEE, 2017.

[43] Kris Kwiatkowski, Panos Kampanakis, Bas Westerbaan, and Douglas Stebila. Post-quantum hybrid ECDHE-MLKEM Key Agreement for TLSv1.3. Internet-Draft draft-kwiatkowski-tls-ecdhe-mlkem-03, Internet Engineering Task Force, December 2024. Work in Progress.

[44] Markus Krabbe Larsen and Carsten Schürmann. Nominal state-separating proofs. Cryptology ePrint Archive, Paper 2025/598, 2025.

[45] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7, 2023.

[46] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1533–1557, 2023.

[47] Nikolaos Makriyannis, Oren Yomtov, and Arik Galansky. Practical key-extraction attacks in leading MPC wallets. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 3053–3064. ACM, 2024.

[48] Nikos Mavrogiannopoulos, Frederik Vercauteren, Vesselin Velichkov, and Bart Preneel. A cross-protocol attack on the TLS protocol. 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 62–72. ACM, 2012.

[49] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.

[50] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a secure messenger. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1289–1306. USENIX Association, 2023.

[51] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1256–1274. IEEE, 2019.

[52] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 983–1002. IEEE, 2020.

[53] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1465–1482. USENIX Association, 2019.

[54] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[55] Mike Rosulek. *The Joy of Cryptography*. 2025. `https://joyofcryptography.com`.

[56] Douglas Stebila, Scott Fluhrer, and Shay Gueron. Hybrid key exchange in TLS 1.3. Internet-Draft draft-ietf-tls-hybrid-design-12, Internet Engineering Task Force, January 2025. Work in Progress.

[57] 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.

[58] The Coq Development Team. The Coq Proof Assistant. 2024.

[59] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. Verifying dynamic trait objects in rust. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 321–330, 2022.

[60] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. Treesync: Authenticated group management for messaging layer security. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 1217–1233. USENIX Association, 2023.

[61] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 564–578. ACM, 2023.

[62] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. 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 1789–1806. ACM, 2017.

[63] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

[64] Sacha Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. A hybrid approach to semi-automated rust verification, 2025.