OwlC: Compiling Security Protocols to Verified, Secure, High-Performance Libraries (Extended Version)

Pratap Singh
Carnegie Mellon University

Joshua Gancher
Northeastern University

Bryan Parno
Carnegie Mellon University

Abstract

Cryptographic security protocols, such as TLS or WireGuard, form the foundation of a secure Internet; hence, a long line of research has shown how to formally verify their high-level designs. Unfortunately, these formal guarantees have not yet reached real-world *implementations* of these protocols, which still rely on testing and ad-hoc manual audits for security and correctness. This gap may be explained, in part, by the substantial performance and/or development overhead imposed by prior efforts to verify implementations.

To make it more practical to deploy verified implementations of security protocols, we present OwlC, the first *fully automated*, security-preserving compiler for verified, *high-performance* implementations of security protocols. From a high-level protocol specification proven computationally secure in the Owl language, OwlC emits an efficient, interoperable, *side-channel resistant* Rust library that is automatically formally verified to be correct.

We produce verified libraries for all previously written Owl protocols, and we also evaluate OwlC on two new verified case studies: WireGuard and Hybrid Public-Key Encryption (HPKE). Our verified implementations interoperate with existing implementations, and their performance matches unverified industrial baselines on end-to-end benchmarks.

1 Introduction

Security protocols, such as TLS, Kerberos, and Signal, are vital components of modern digital infrastructure, but vulnerabilities are discovered in them with alarming frequency [1, 14, 68, 71]. Attacks on these critical protocols—both against high-level design flaws, and low-level implementation details—can have outsized impacts.

Formal verification promises to provide a foundational solution to this problem, by developing rigorous machine-checked proofs of security. Extensive prior work has verified the designs of cryptographic protocols, leading to a range of tools [8, 10, 15, 21, 42, 67]. While such tools can give powerful assurance about the logical structure of protocols, they elide many details that must appear in actual implementations. The code that runs when a protocol is ultimately deployed thus has only a tenuous, informal connection to the computer-aided proof. As we discuss in §11, additional work has sought to verify implementations of protocols, using a range of approaches. Some tools have translated verified protocol designs into unverified implementations [16, 42, 56] or specifications suitable for manual program verification [7]; others [18, 33, 49] have, with heroic effort, manually verified a single protocol or family of protocols. However, none can verify both the design and a performant implementation of a protocol in an automated, reusable fashion.

We present OwlC, the first tool that can automatically generate verified, interoperable, side-channel-resistant implementations of computationally secure cryptographic protocols. Given a protocol design, OwlC generates a high-performance memory-safe Rust library that is automatically *verified* to preserve the security properties of the design. In particular, OwlC guarantees that the implementation does not leak any data that is secret in the protocol design.

OwlC is based on a novel, lightweight technique for building *proof-producing compilers* for domain-specific languages with effects. Our approach features two components: a simple, trusted specification compiler that translates the source program into specification code in the target verification language, and an untrusted compiler that generates efficient, verified, executable code. Our security-preservation technique is based on restricting the externally observable behavior of the generated code, encompassing both explicit leakage via input-output behavior and implicit leakage via digital side-channel attacks. To prevent explicit leakage, we compile specifications as *interaction trees* [92], a higher-order data structure consisting of effects and their continuations. We then interpret these interaction trees as *permissions to perform input-output effects* in executable code; thus if verification succeeds, we prove that the executable code performs exactly the network I/O that the protocol specified.

To prevent implicit leakage (e.g., through timing), we guarantee resistance to basic digital side-channel attacks via a type abstraction mechanism similar to prior work on verified cryptographic primitives [95]. However, our new focus on protocols requires a new account of *declassification*, which historically has been encoded through ad-hoc trusted APIs. Instead, we leverage our interaction tree-based specifications to specify a semantically meaningful *policy* on declassification that applies to all protocols automatically.

OwlC instantiates this technique by compiling protocol designs in the Owl protocol language [42]. Owl is a state-of-the-art protocol design verifier that provides scalability, automation, and strong cryptographic security results. We choose Owl as our source since—unlike many prior tools—Owl's input language is designed to be compatible with real-world concerns, such as network packet formats. Owl's information-flow types also tell us which values can be made public, which we use in our declassification technique.

OwlC compiles Owl protocols to Rust [55] libraries, which we verify using Verus [61], an SMT-based deductive program verifier for Rust. By carefully employing Verus's powerful proof automation and support for ergonomic permission tracking, OwlC's libraries typically compile and verify *fully automatically*, without any user-specified annotations. Since the compiled code is in safe Rust, it is also automatically memory-safe. Application developers can seamlessly integrate OwlC's generated libraries into a larger Rust codebase using `cargo`. We also illustrate how verified applications can use our protocol specifications in the service of a larger correctness or security guarantee (§7), facilitating the development of end-to-end verified applications.

We demonstrate the utility and scalability of OwlC via several case studies. We compile verified libraries for all 14 protocol case studies from the original Owl work [42], including the core logic of SSH and Kerberos; however, these case studies made a number of simplifications relative to realistic implementations. To show that OwlC can automatically generate performant, interoperable implementations of real-world protocols, we perform two large-scale case studies on **(1)** WireGuard [36], a widely-adopted VPN protocol based on state-of-the-art cryptography, and **(2)** HPKE [12], a recent standard for hybrid public-key encryption. We implement and prove cryptographic security for WireGuard and HPKE in Owl, and then we use OwlC to compile verified libraries. Our generated code interoperates with existing implementations of WireGuard and HPKE, and achieves state-of-the-art performance on end-to-end benchmarks.

Thus, OwlC eliminates the difficult task of verifying the connection between an implementation of a protocol and its specification. Protocol designers using OwlC can obtain for free a performant, interoperable library for their protocol, plus a proof that this library is functionally correct, memory safe, and free of both explicit and implicit leakages.

Limitations. OwlC's guarantees rely on the correctness of the underlying tools, namely Owl, Verus, and Rust. We trust the correctness of the simple specification compiler that generates interaction trees from Owl protocols. While our type abstraction provides resistance to timing- and memory-based side-channel attacks at the level of Rust source code, we do not provide assembly- or hardware-level guarantees since we use stock `rustc` to compile OwlC's output.

Contributions. In summary, this paper contributes:

1. A technique for developing proof-producing compilers for domain-specific programming languages with effects, based on higher-order ghost linear tokens enforcing permissions to perform effects.
2. A technique for providing source-level digital side-channel protections, combining type abstraction with a semantically founded policy for declassification, the first such technique for cryptographic protocol code.
3. OwlC, an instantiation of these techniques and the first tool that automatically produces verified, performant, interoperable, and side-channel-resistant libraries from computationally secure protocols specified in a high-level language.
4. Over a dozen compiled protocol case studies, plus the first automatically verified, interoperable implementations of the HPKE and WireGuard protocols with the performance of unverified industrial implementations.

2 Background and Threat Model

To verify protocols for cryptographic security, OwlC utilizes Owl [42], a verification framework for security protocols. Owl uses an intuitive, ML-style source language equipped with an *information-flow type system* that guarantees *computational security* [22], the cryptographic "gold standard" which demonstrates that the protocol is secure against arbitrary probabilistic attackers who may perform arbitrary computations on low-level bitstrings. OwlC then translates well-typed Owl protocols into provably secure Rust code.

To produce and analyze verified implementations, we use Verus [60, 61], a state-of-the-art program verifier for Rust. Verus verifies Rust programs using annotations for pre- and post-conditions, loop invariants, and inline assertions, all of which generate verification conditions that are checked by the Z3 [31] SMT solver. Verus also takes advantage of Rust's strong ownership type system to dramatically simplify reasoning about mutable memory.

Verus further leverages Rust's type system to support *linear ghost state*, wherein ghost variables (i.e., variables used only for proof purposes, not compiled code) are subject to the ownership rules of Rust's type system. In essence, linear ghost state allows Verus developers to enforce complex program invariants that *evolve* during execution.

2.1 Threat Model and Security Guarantee

We inherit Owl's security guarantee, which proves that no arbitrary polynomial-time cryptographic adversary interacting

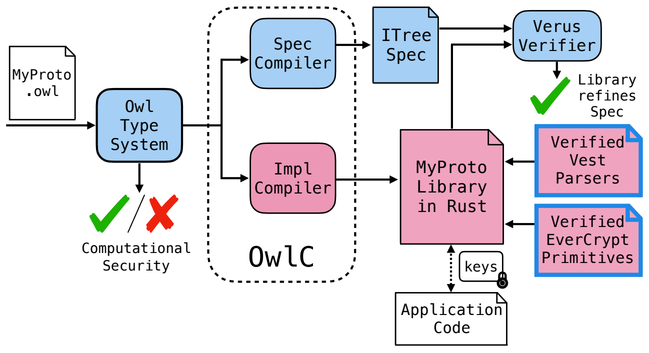

Figure 1: **Architecture of OwlC**. *Blue elements are trusted components, while red elements are untrusted; a blue border indicates a verified component whose specification is trusted by OwlC.*

with the protocol's functional specification can violate its type-based security specification.

In turn, OwlC guarantees that compiling this functional specification to realistic low-level code is *security preserving*: any low-level adversary interacting with the compiled Rust code is *equivalent* to a high-level one interacting with the functional specification. For OwlC, low-level adversaries are those which may interact with compiled Rust modules in type-safe ways (e.g., receiving and delivering untrusted network messages, and calling functions at API boundaries) and observe digital side channels, such as memory access patterns and timing.

To guarantee that compilation is security preserving, we prove two key properties: *functional correctness*, including performing *only* the I/O effects dictated by the source program; and *side-channel resistance*, meaning that the untrusted output of our compiler cannot exfiltrate secrets through observable timing or memory behaviors. Further side channels, including microarchitectural channels, are out of scope.

Trusted Computing Base. Our TCB consists of the Rust compiler, Verus, Owl, and OwlC's specification compiler. While we statically forbid unsafe Rust in OwlC's output, bugs in our TCB could compromise memory safety or our security-preservation guarantee. OwlC relies on an existing verified library of cryptographic primitives [79], but we axiomatize the connection between OwlC's output and this library (§8.1). For side-channel security, we trust that our secret type is implemented in constant time (§5.1).

3 Overview of OwlC

Given a security protocol written in the Owl language, OwlC generates a corresponding verified, high-performance implementation. In this section, we summarize OwlC's architecture and verification techniques.

Our Approach: Translation Validation. OwlC compiles Owl protocols to Rust code formally verified using the Verus framework. Verus enables the creation of verified software that is uncompromising in both *correctness* and *performance*, both of which are essential for security protocols.

To use Verus to generate verified implementations, we do not verify the correctness of the compiler itself (e.g., as in CompCert [62] or CakeML [59]); instead, inspired by translation validation [70, 72, 78], we generate proofs of correctness and security in Verus *alongside* the implementation. Translation validation allows us to verify implementations without conducting proofs that universally quantify over programs.

Figure 1 outlines our approach. Given an Owl protocol, we first type check it using Owl to ensure cryptographic security. Then, OwlC translates it into two representations: a Verus specification, to embed the high-level semantics of the Owl program into Verus; and a library of executable implementation routines, which utilize mutable state and efficient cryptographic operations to run the protocol. The translation to specifications is trusted for correctness, while the translation to libraries is not. Additionally, to provide high-assurance implementations, OwlC uses two other verified components: Vest [27], a library of verified combinators for reading and writing network formats; and EverCrypt [79], a library of verified cryptographic implementations. Finally, developers can use the generated Verus library to build a larger verified application; OwlC guarantees via the Rust type system that protocol secrets and derived keys are securely encapsulated when returned to application code.

Preventing Explicit Implementation Leakage. OwlC must guarantee that the generated code performs only actions that have been proved secure by the Owl type-checker. Hence, it must prevent any input-output operations that do not appear in the Owl program, since such operations—such as writing a secret key to the network, or outputting extra values—could compromise the protocol's security. To do so, OwlC encodes specifications using Interaction Trees (or ITrees [92]), an embedding of effects into a verifier. We then use the generated ITree specifications, combined with Verus's support for linear ghost permission reasoning, to control input-output operations in the generated library (§4).

Preventing Implicit Implementation Leakage. Even in the absence of explicit leakage, a protocol implementation might implicitly leak secrets via side-channels. OwlC prevents this using a type abstraction mechanism, in which secret data is encapsulated in an opaque type that only allows constant-time operations. Type abstraction is a well-established technique that has been especially successful for verified implementations of cryptographic primitives [95]; in that setting, certain inputs are designated as secret once and for all, so it is straightforward to add opaque types. Protocols, on the other hand, may require explicit *declassification* of opaque secrets into public values; for instance, protocols may branch on whether a decryption operation succeeds, even if the decrypted plaintext remains secret. However, allowing arbitrary declassifications clearly compromises any guarantee of side-channel security. OwlC therefore carefully uses Verus's linear ghost permissions to control access to declassification, guaranteeing that any value that is declassified in the implementation is always public in the Owl protocol (§5).

4 Restricting Explicit Input-Output Leakages

In this section, we explain our approach to preventing explicit information leakage by restricting the input-output behavior of the generated library. §4.1 explains how we represent the input/output behavior of Owl protocols as Verus specifications. §4.2 shows how we verify that compiled Owl code adheres to its input/output specification, as well as how that verification technique generalizes.

```
 1 locality Alice, Bob
 2 name kA2B, kB2A : enckey
 3
 4 struct EncMsg {
 5    version_num: Const(0x01)
 6    cipher : Bytes⟨public⟩
 7 }
 8
 9 def alice_send(m : Bytes⟨secret⟩) @ Alice {
10    let send_key = get(kA2B) in
11    let c = enc(send_key, m) in
12    output EncMsg(0x01, c)
13 }
14
15 def alice_recv() @ Alice : Option Bytes⟨secret⟩ {
16    input p in
17    parse p as EncMsg(_, c) in {
18        let recv_key = get(kB2A) in
19        dec(recv_key, c)
20    }
21    otherwise { None() }
22 }
```

Figure 2: **Basic Secure Transport in Owl**. *Details irrelevant for compilation have been erased.*

Example: Secure Transport. We begin by introducing a simple running example. Figure 2 shows a fragment of a simple Owl protocol for secure transport, where Alice and Bob communicate over a previously established secure channel, constructed using two unidirectional keys. To simplify the presentation, we show the protocol after OwlC's *concretization* pass (Appendix D); hence, Owl's fine-grained information-flow types have been flattened to just secret and public, corresponding to values that are possibly secret or always public, respectively.

To implement the secure channel in Owl, we first in Line 1 specify the two *localities*, or parties, which serve to organize the protocol into independent pieces of executable code. Next, Line 2 declares *names*, which are the cryptographic secrets

```
enum ITree⟨A⟩ {
  Ret(A),
  Input(Seq⟨u8⟩ → ITree⟨A⟩),
  Output(Seq⟨u8⟩, ITree⟨A⟩),
  Sample(usize, Seq⟨u8⟩ → ITree⟨A⟩),
  Declassify (DeclassifyingOp, ITree⟨A⟩)
}

fn bind⟨A, B⟩(t : ITree⟨A⟩, k : A → ITree⟨B⟩) → ITree⟨B⟩ { ... }
```

Figure 3: **Definition of the ITree Data Structure in Simplified Verus Syntax**. *The* bind *operator is used to implement sequential composition of ITrees.*

that the protocol uses. The two names kA2B and kB2A represent encryption keys for sending messages from Alice to Bob, and vice versa. The annotation enckey declares that both of these keys should be sampled according to the underlying symmetric encryption scheme.

The struct EncMsg on Lines 4-7 specifies the network format for the secure channel, consisting of a protocol version number followed by the ciphertext. The version field has type Const(0x01), specifying the single byte 0x01.

Finally, Lines 9-22 contain the routines for Alice to send and receive encrypted messages. The routine alice_send obtains the sending key kA2B via a call to get (Owl's abstraction for looking up pre-configured local keys), encrypts the message m, and outputs the version number and ciphertext on the network. For simplicity, encryption is probabilistic and involves randomly sampling an appropriate nonce; OwlC also supports *counter*-based nonces, which we use for WireGuard and HPKE in §9.

The routine for alice_recv is similar, but it requires parsing a message p from the network into the EncMsg struct. Parsing may fail (e.g., if p is too short or does not begin with the constant 0x01); in this case, we must provide an otherwise clause that handles this failure explicitly.

4.1 ITrees for Specifying Effectful Code

The core of our verification technique is to use *interaction trees* (or ITrees [92]) to guarantee that untrusted protocol code generated by OwlC not only returns the correct values, but also performs *exactly* the side effects (e.g., network I/O) dictated by its specification, *and nothing more*. The latter is critical, since a spurious output of a key on the network would destroy the protocol's security.

To specify code that uses side effects, ITrees allow one to embed effectful code inside a proof assistant; they explicitly represent effects through datatype constructors that hold *continuations*, i.e., functions that define how to handle the effects. As shown in Figure 3, our ITree⟨A⟩ enum has five constructors. The Ret constructor injects values of type A into the ITree, while Input, Output, and Sample perform network I/O

```
spec fn alice_recv_spec(cfg: AliceCfg, state: AliceSt)
  → ITree⟨(Option⟨Seq⟨u8⟩⟩, state_Alice)⟩ {
  owl_spec! (
    (input (p)) in
    (parse (p) as (EncMsg { _, ctxt }) in {
      let recv_key = (ret(cfg.kB2A)) in
      (declassify(ADec(recv_key, ctxt))) in
      (ret(Some(dec_spec(recv_key, ctxt))))
    } otherwise (ret(None())))
  )
}
// the owl_spec! macro above expands to:
Input(|p|
  if let EncMsg(_, c) = parse_EncMsg_spec(p) {
    Declassify(ADec(cfg.kB2A, c),
      Ret((Some(dec_spec(cfg.kB2A, c)), state)))
  } else {
    Ret((None, state))
  })
```

Figure 4: **ITree for Alice's Receive Routine in Secure Transport**. *OwlC generates the* owl_spec! *macro invocation, which Rust internally expands to the ITree shown below.*

and sampling from a random source. The final constructor, Declassify, is used for side-channel resistance (see §5). Each effect constructor contains an argument as well as a *continuation* that describes the ITree's behavior after performing the effect. Additionally, ITrees support sequential composition via a monadic *bind* operator; this is useful for compositionally reasoning about ITrees.

Our implementation of ITrees in Verus uses an inductive type (meaning any value with that type has a finite size), as opposed to ITrees in Coq [92] which are coinductive; thus, our ITrees only handle terminating computations and cannot handle general loops. This is sufficient for OwlC, since we encode long-running protocols via terminating message handlers that may be called multiple times.

Using ITrees, we can automatically translate Owl protocols into Verus specs through a simple compilation pass. Figure 4 shows an ITree-producing function (alice_recv_spec) for Alice's receive routine from Figure 2. Notice the nearly line-for-line correspondence with the original Owl routine. This correspondence simplies our trusted spec compiler, and it makes manual audit feasible, if desired. We achieve this close corresponce via a 119-line Rust macro *owl_spec!* that expands to the ITree constructors shown in the bottom half of Figure 4.

Compared to the original Owl routine, OwlC makes small syntactic modifications, chiefly adding parentheses to make scoping explicit, to account for Rust's macro parser. The spec function (and the ITree it produces) is parameterized by two pieces of state: cfg, which holds Alice's keys that do not change throughout the protocol; and state, which holds mutable counters for stateful AEAD (Appendix A), not used in this example. Decrypting the ciphertext requires declassification (§5.2), which we mark explicitly in the ITree specification.

4.2 Linear Ghost State for Controlling Effects

Historically, ITrees have been used to give denotational semantics to programming languages [54, 85, 94]. By themselves, they do not provide a means for *controlling* how those effects are performed in executable code. In OwlC, our key idea is to require the implementation to thread through a ghost linear *token* for the ITree specification of an Owl routine; the token grants the implementation permission to perform the ITree's I/O effects.

For example, if the implementation wishes to perform an input and currently holds token T, we require that T is of the form Input(|x| k(x)). If it is, then the implementation may "trade in" T to perform the input. In return, the implementation receives a message, v, from the network along with the token k(v), which allows it to perform further effects. We use Rust's type system to ensure that tokens are unforgeable and cannot be duplicated, so the implementation may only perform effects according to its ITree token.

We encode the above interface using Verus's support for linearity, abstract types, and verification conditions. In particular, our definition of ITree tokens and their interface is given in Figure 5. The token, ITreeToken⟨A⟩, holds a ghost ITree that returns a value of type A. Importantly, since the inner field of the token is private, and the token does not implement Clone or Copy, the linearity and scoping rules of Rust ensure that tokens cannot be copied, duplicated, or constructed. In addition to the API in Figure 5, we also support functions for *splitting* a monadic bind, bind(t, k), into two tokens for t and k, and later *joining* a monadic return Ret(v) and continuation k back together to form a token for k(v); these two functions are crucial for supporting modular verification of subroutines.

To perform network I/O, the implementation must use input and output, given on Lines 5 and 11, respectively; we ensure syntactically that no other I/O routines are accessible to the implementation. The input function formalizes the informal description above, while output requires that the provided token is of the form Output(v, k), and that the value to be output actually is v; in return, output modifies the token to hold k. The function sample is constructed similarly and internally calls a cryptographically secure PRNG.

These functions are *trusted* (indicated by *#[axiom]*). The verifier cannot reason about the details of the various system calls implementing the network and sampling routines. Instead, these axiomatized specifications can be thought of as defining for the verifier what it means to perform an effect.

Using Tokens in Executable Code. Figure 6 shows an example of how OwlC compiles and verifies code using ghost linear ITree tokens. OwlC compiles an Owl procedure to an executable Verus function, which takes as an argument a mutable reference to a linear ghost ITree. We use a **requires** clause to demand that this reference initially points to the ITree representation of the Owl procedure generated by the specification compiler. We add an **ensures** clause stating that

```
mod ITreeToken {
    pub struct ITreeToken⟨A⟩(inner : Ghost⟨ITree⟨A⟩⟩);

    #[axiom]
    pub fn input⟨A⟩(t : &mut ITreeToken⟨A⟩) → (v : Vec⟨u8⟩)
        requires old(t).inner is Input
        ensures t.inner == old(t).inner.continuation(v)
    { /* receive v from network */ }

    #[axiom]
    pub fn output⟨A⟩(t : &mut ITreeToken⟨A⟩, v : &[u8])
        requires old(t).inner is Output,
                 old(t).inner.output_value == v
        ensures t.inner == old(t).inner.continuation
    { /* output v on network */ }

    // sample is similar
} // ITreeToken is unforgeable in this scope
```

Figure 5: **Interface for ITree Token and Associated Executable Effects in Verus**. *(Simplified)*

```
fn go_spec() -> ITree<u8> { Input(|x| Output(f(x), Ret(42))) }
```

fn go(&mut tok) -> (res : u8) requires tok == go_spec() ensures tok == Ret(res)	Program State	I/O Events
	tok = Input(|x| Output(f(x), Ret(42)))	
let x = input(&mut tok);	x = v	input v
	tok = Output(f(v), Ret(42))	

let y = f(x);
output(&mut tok, y);
		output f(v)

42	tok = Ret(42)	

Figure 6: **Controlling the Evolution of Program State Using Tokens**. *The code on the left must follow the effects defined by go_spec, using the ghost linear argument* `tok`.

after execution, the ITree should be of the form Ret(res), where res is the result of the executable function. This serves two purposes: it guarantees that the compiled code performs *all* of the effects specified by the Owl procedure, rather than just some prefix of them; and it shows that the compiled code computes and returns the correct value.

In general, translation validation requires the compiler to emit proof hints that help the verifier prove that the generated implementation refines its specification. OwlC does not need to emit any such hints: verification proceeds *completely automatically* just using the ITree specification and the executable code. This is primarily due to our use of linearity: the verifier can reason straightforwardly about the ITree specification and the axioms for the effects (Figure 5), without having to prove expensive verification conditions about mutable memory or token unforgeability.

Generalizing Our Verification Technique. While our verification technique is useful for OwlC, it is broadly useful for verifying software that performs observable effects. Indeed, almost any effect with the signature effect : Inputs → Outputs is expressible in our framework; the only restriction is that the continuation for the effect must be used *linearly*, to soundly interoperate with executable code in Rust.

Our technique can also be generalized to target verification tools besides Verus. OwlC relies on datatype abstraction and linearity to ensure that the ITree token cannot be duplicated or copied. Any other program verification tool that natively supports substructural reasoning (e.g., Iris [53] or Steel [41]) could also encode ITree tokens. In the absence of linearity, one could manually enforce that tokens cannot be duplicated or copied using verification conditions [46].

5 Restricting Implicit Side-Channel Leakages

The techniques described in §4 guarantee that OwlC's generated Rust code is secure against logical and cryptographic attacks; however, protocol code must also be secure against lower-level side-channel attacks. In particular, protocol implementations that satisfy the secrecy guarantees provided by Owl may still leak information about secrets via their timing and memory behavior. For example, the Raccoon attack [81] against TLS 1.2 relies on hash functions leaking information about their input lengths via timing: since TLS 1.2 requires stripping leading zeroes from some secret values before hashing, precise timing measurements can leak the secret's number of leading zeroes.

To our knowledge, ours is the first systematic technique for automatically providing side-channel resistance to cryptographic protocol implementations. Our use of type abstraction (§5.1) to prevent secrets from influencing control flow follows an established technique, but our approach to restricting declassification to only provably public data (§5.2) is novel, and critical to the security guarantees of OwlC.

5.1 Constant-Time via Type Abstraction

Similar to prior work [95], OwlC guarantees adherence to a cryptographic constant-time coding discipline [2, 3, 13] via type abstraction, which entails encapsulating secret values within opaque wrapper types that prevent code from observing or branching on them. However, unlike prior work, we target *protocols*, rather than *primitives*, necessitating a more expressive security policy.

Our opaque wrapper type is SecretBuf, shown in Figure 7, which hides a byte buffer containing a secret value, exposing an API that only allows operations that can be implemented in constant time. SecretBuf is built on top of PublicBuf, a OwlC-specific type for byte buffers that we introduce for performance reasons (described in §6). Note that while the SecretBuf struct itself is **pub**, the inner buf field is private, meaning that it cannot be accessed outside of the secret module. Each of the four API functions is implemented with simple straight-line Rust code, with no branching that could leak information about the contents of the secret buffer.

```
pub mod secret {
  pub struct SecretBuf { buf: PublicBuf }
  impl SecretBuf {
    pub fn len(self) → usize { ... }
    pub fn from_public(buf: PublicBuf) → SecretBuf { ... }
    pub fn subrange(self, start: usize, end: usize) → SecretBuf { ... }
    pub fn concat(self, other: SecretBuf) → SecretBuf { ... }
  }
}
```

Figure 7: **SecretBuf Definition and API**. *We elide some Verus details such as lifetimes and requires/ensures specifications.*

To demonstrate SecretBuf's utility, we attempted to write a function to strip leading zeroes from a SecretBuf as in the Raccoon attack, but Rust's typechecker immediately rejected it—there is no way to read the encapsulated bytes.

Limitations. Our digital side-channel resistance property is only enforced at the level of Rust source programs, so we must trust that `rustc` does not introduce any new side-channel leakages. While such an assumption may be unrealistic for stock `rustc`, prior work has developed constant-time-preserving compilers [13, 28], so we may expect to compile Rust in this way in the future.

5.2 Handling Semantic Declassifications

Side-channel resistance for cryptographic protocols is significantly more complex than for primitives, since non-trivial protocols are *not* classically constant time. For example, protocols may branch on whether a decryption $\mathsf{dec}(k, c)$ succeeds, even if k is a secret. In our setting, Owl proves that it is safe to branch on whether a decryption succeeded; intuitively, this is because with an authenticated encryption scheme, the adversary already knows that ciphertexts produced by legitimate parties will successfully authenticate, while those the adversary produces will fail. Verifying side-channel resistance for protocols therefore requires reasoning about *semantic declassifications*, in which a secret value (such as the result of decryption) becomes public when justified by the cryptographic invariants of the protocol. Importantly, such declassifications do not appear when verifying primitives (e.g., ciphers). Further, OwlC's generated library code should not be allowed to declassify arbitrary values, since a faulty or malicious implementation could then declassify a secret, potentially resulting in side-channel leakage.

Our solution is based on the observation that Owl already computes which values are safe to declassify—Owl's type system features a fine-grained information-flow lattice that considers all possible corruption scenarios for the names in the protocol. We can then justify declassifications in the generated code by proving that Owl assigns a public type to every declassified value. To model this, we insert *declassifications* as effects in the ITree (Figure 3).

```
pub enum DeclassifyingOp {
  ControlFlow (Seq⟨u8⟩),
  EnumParse (Seq⟨u8⟩),
  EqCheck (Seq⟨u8⟩, Seq⟨u8⟩),
  ADec (Seq⟨u8⟩, Seq⟨u8⟩), // key, ciphertext
  ... // other crypto ops that may fail
}
```

Figure 8: **DeclassifyingOp Definition**.

```
mod DeclassifyingOpToken {
  pub struct DeclassifyingOpToken(inner: Ghost⟨DeclassifyingOp⟩);
} // DeclassifyingOpToken is unforgeable in this scope

#[axiom]
pub ghost fn consume_itree_declassify⟨A⟩(t: &mut ITreeToken⟨A⟩)
  → (d: DeclassifyingOpToken)
  requires old(t).inner = Declassify(op, continuation)
  ensures t.inner == continuation,
              d.inner == op
{ /∗ no−op ∗/ }

#[axiom]
pub fn declassify(s: SecretBuf, d: DeclassifyingOpToken)
  → (res: PublicBuf)
  requires d.inner == ControlFlow(s)
  ensures res == s.buf
{ /∗ return contents of s ∗/ }
```

Figure 9: **Declassification in Implementation Code**.

Unlike other effects which can only happen through Owl primitives (input/output/sample), declassification can occur in a number of places in an Owl program: these include branching control flow; parsing bytes as an **enum**, which declassifies the first tag byte; checking equality of two secrets; and using cryptographic primitives that may fail. We encode these possibilities via our DeclassifyingOp type (Figure 8), which is passed as an argument to Declassify nodes in the ITree.

For declassification in executable code, one might imagine a single declassify primitive that takes an ITreeToken and the arguments to be declassified, similar to the input and output primitives in Figure 5. However, the ITree enforces an ordering of all effects, including declassification; while this is desirable for I/O, OwlC's optimizations may re-order some cryptographic operations for greater efficiency (§6). Thus, we instead split this routine into two, as shown in Figure 9. First, consume_itree_declassify consumes the Declassify node in the ITree to produce a DeclassifyingOpToken, which serves as a ghost linear permission to perform a declassification later on. Then, in turn, declassify consumes this declassification token to turn a corresponding SecretBuf into a PublicBuf. Here, the declassification corresponds to control flow (i.e., branching on a value, which reveals the branch); other routines for parsing into **enum**s, checking equality of secrets, and unwrapping cryptographic operations are similar.

Inserting Declassification During Compilation. The first order of business for OwlC is to determine which Owl values should be in SecretBufs and which can be public. OwlC begins by collapsing Owl's information-flow lattice to two levels: always-public, for values that are always safe to leak to the adversary; and possibly-secret-with-public-length, for values that may in some scenarios be unsafe to leak. The always-public values are compiled into PublicBufs, while the possibly-secret values are compiled into SecretBufs. Importantly, Owl's type system already guarantees that only public values can drive control flow [42]. Next, OwlC inserts declassifications in cases where Owl deems a value to be always public, but that value is held in a SecretBuf. For instance, if a protocol decrypts a public constant, the decryption routine will put the plaintext in a SecretBuf, but it should be treated as public. In this case, OwlC will insert a ControlFlow DeclassifyingOp into the ITree, plus calls to consume_itree_declassify and declassify in the implementation code.

Public Lengths of SecretBufs. Note that Figure 7 allows unrestricted access to the length of the SecretBuf. This is required for operations such as serialization and encryption/decryption, since we may need to allocate a fresh buffer for the output; the length of the output buffer depends on the length of the input SecretBuf. The original Owl type system [42] allowed types to have secret lengths; since we cannot express such types in Verus, we modified the typing rules of Owl to additionally forbid constructing values with secret lengths. We found that doing so did not prevent us from modeling and verifying realistic protocols in Owl, indicating that protocols do not typically use data with secret lengths. Indeed, operations over secret-length data are prone to side-channel leakage; for example, hashing a buffer with a secret length will likely introduce a timing channel [81].

6 Generating Fast, Interoperable Libraries

We now turn to compiling safe, efficient, and interoperable libraries using only the information in the Owl source program. Protocols are particularly suitable for automated code generation: they typically consist of relatively short procedures with straightforward control flow, weaving together various building blocks, such as cryptographic operations, parsing, and network communication. While a naïve compilation is simple, obtaining competitive performance involves several subtleties. We defer a full description of the compiler's implementation to Appendix D, focusing here on how we address three key challenges.

Type-Guided Compilation. OwlC does not need to formally prove that its emitted code is valid Rust, since Verus type checks it. Nevertheless, we do need to track enough Rust-level detail so that our emitted code does not violate the Rust type system. Rust guarantees memory safety via ownership types and a borrow checker for *lifetimes*, abstract symbolic constraints on how owned values can be borrowed.

A naïve solution to satisfy Rust's borrow checker would be to copy each value every time it is used; however, this would be prohibitively slow. Instead, we note that protocols largely operate on byte buffers, and focus on generating efficient code to manipulate these buffers. We use the type PublicBuf, a data structure that efficiently supports immutable operations on byte buffers. PublicBuf internally uses reference counting, so that PublicBuf values can be copied cheaply *without* copying the underlying bytes in memory. PublicBuf allows OwlC to compile functional Owl protocols to efficient Rust code, without detailed reasoning about lifetimes in the compiler.

Leveraging Verified Parsing. To transmit structured data over the network, protocols rely on serializers and parsers, which present particularly large attack surfaces, since they manipulate low-level buffers and directly handle adversarial inputs. Buffer overflows from reading untrusted data can have catastrophic effects [29, 71], while format confusion attacks can trick implementations into holding false beliefs [65, 90].

To minimize this attack surface, we leverage Vest [27], an existing Rust library of *verified, bidirectional* parser/serializer combinators for Verus, which eliminates large classes of format confusion and parser malleability attacks [80, 88, 90]. Vest also provides side-channel security via type abstraction, so we can map OwlC's type-protected secrets to Vest's.

Fused Operations. Efficient protocol code must (where possible) mutate memory in-place and avoid excessive allocations. However, Owl is a high-level functional language, so a naïve translation of Owl into Rust might allocate a fresh buffer for every intermediate value, imposing significant runtime costs.

To avoid these costs, OwlC lazily evaluates certain computations in order to *fuse* them into their final destination buffer, once it is known. For example, if the code encrypts, serializes, and then outputs a value, all of those operations can be fused into a single computation. This computation mutates a single buffer in place, avoiding the allocation of intermediate buffers for the ciphertext and serialized datatype.

OwlC currently supports a set of fuseable operations for performance-critical code, including cryptographic operations (e.g., encryption) and network output. We also extend Vest to support fused operations, whereby computations (including cryptographic operations) may happen inside the serialization routine for a given datatype. Importantly, the details of these optimizations are untrusted—they do not appear in specifications and are verified to be functionally correct.

7 Using OwlC's Protocol Libraries

While OwlC focuses on verifying protocol implementations, these protocols may be deployed in larger verified systems. Hence, we illustrate how such systems can use OwlC's generated libraries and formal specs.

Broadly, there are two possible approaches. The most cryptographically sound approach is to implement the application logic in Owl, and prove secrecy and integrity for the whole application using Owl's information-flow type system. The developer can then use OwlC to generate a verified Rust library for the application logic. This approach enables end-to-end cryptographic proofs for the composed system. However, Owl may be awkward for certain applications due to its high-level representation of protocols and current limitations (e.g., the lack of recursion).

Alternatively, a developer can write the application logic directly in Verus, with a handwritten ITree for its spec. This handwritten ITree can call OwlC's generated ITrees to invoke protocol functionality. ITrees can be recursive, so the developer can naturally express applications with repeating behavior. However, Verus does not (presently) feature information-flow control or cryptographic reasoning, so this approach cannot prove properties beyond functional correctness without significant extensions to Verus.

Echo Server. To illustrate these approaches, we have used each to implement a basic secure echo server: upon receiving a message from a client, the echo server simply sends the same message back to the client. Messages are transmitted using the secure transport example from Figure 4; for simplicity, we assume the server has been pre-configured with sending and receiving keys (the names kA2B and kB2A from Figure 2).

7.1 Implementing Application Logic in Owl

```
def echo_server() @ Server : Unit =
  let recv_result = call alice_recv() in
  case recv_result {
    | Ok ptxt ⟹ call alice_send(ptxt)
    | Err ⟹ ()
  }
```

Figure 10: **Echo Server Implementation in Owl**. *The functions alice_recv and alice_send are defined in Figure 2.*

Figure 10 shows our Owl code for the echo server, which straightforwardly encodes the application logic. Owl proves via typing that the plaintext ptxt remains secret from the adversary, and that unless the server's keys are corrupted, the client receives back an encryption of the same ptxt it sent.

Note that Owl does not yet allow recursive definitions or loops, so echo_server only runs the echo server once. However, it is sound to run echo_server an arbitrary number of times; hence, we write a trusted wrapper that just calls the compiled Rust routine for echo_server in an infinite loop. Since this is a trivial 3-line loop, it is straightforward to audit for correctness.

7.2 Implementing Application Logic in Verus

To write the verified application in Verus, we must connect the app's spec to the generated library's spec. We thus specify the application logic as an ITree, using our *owl_spec!* macro.

```
spec fn echo_server(cfg: cfg_Server, fuel: usize) → ITree⟨()⟩ {
  owl_spec! (
    if (fuel == 0) { ret(()) } else {
      let recv_result = (call(alice_recv(cfg))) in
      let _ = (case (recv_result) {
        | Ok(ptxt) ⟹ { (call(alice_send(cfg, ptxt))) },
        | Err() ⟹ { ret(()) },
      }) in (call(echo_server(cfg, fuel − 1)))
    }) }
```

Figure 11: **Handwritten ITree for Echo Server**.

Figure 11 shows our ITree-based spec for the echo server. Because we write it directly in Verus (unconstrained by Owl's limitations), we can make the spec recursive: the echo_server function calls itself to proceed with the next round of the protocol. Since Verus requires all spec functions to terminate, we provide a fuel parameter to bound the recursion. The ITree seamlessly composes with the OwlC-generated ITrees produced by the alice_send and alice_recv spec functions.

On the implementation side, we wrote a loop that calls Alice's compiled send and receive routines. Like similar tools, Verus requires loop invariants to verify loops. We straightforwardly use echo_server to provide the loop invariant: at each iteration, the current ITreeToken should contain the ITree given by echo_server(cfg, cur_fuel), where cur_fuel is the current fuel. Thus, ITrees provide a flexible spec format for verified applications, without constraining implementation choices—we are free to use a more performant looping implementation, rather than perfectly mirroring the ITree's control flow.

8 Implementation of OwlC

We have implemented OwlC as a tool in Haskell, based on the Owl's open-source Haskell implementation. Our two compilers (§3) are 5,934 lines of Haskell, which breaks down as 753 lines for the trusted specification compiler, 2,083 lines for the untrusted implementation compiler, and 3,098 lines of shared definitions and utilities. We also have 1,865 lines of handwritten Verus code for ancillary definitions (§8.1).

Modifications to the Owl Type System. To automatically verify and compile realistic protocols, OwlC extends the Owl type system in a few ways. First, as discussed in §5.2, we restrict Owl's information-flow typing rules to enforce extra constraints for side-channel resistance. Additionally, we support two new cryptographic primitives — *plain-model key derivation functions* and *authenticated encryption with additional data* — as well as *singleton types* which are used to encode magic constants. For more details, see Appendix A.

8.1 OwlC Output

OwlC generates Verus files containing the specifications and implementations of the protocol routines. These files link against trusted handwritten Verus files defining the ITree data

```
#[axiom]
spec fn enc_spec(k, x, nonce, aad : Seq⟨u8⟩) → (c: Seq⟨u8⟩);
#[axiom]
exec fn enc(k, x, nonce, aad : SecretBuf) → (cipher: PublicBuf)
    ensures cipher == enc_spec(k, x, nonce, aad)
{ /∗ call EverCrypt AEAD ∗/ }
```

Figure 12: **OwlC's Embedding of Authenticated Encryption**. *Owl assumes that* enc_spec *models a secure authenticated encryption algorithm. Other primitives are similar.*

structure, the effect functions, and our interface to the cryptographic provider. The result is a verified Rust library crate implementing the protocol. We briefly discuss the crate's relevant external dependencies and its interface.

8.1.1 External Dependencies

Cryptographic Primitives. As a design-level verification tool, Owl does not explicitly model the internals of cryptographic primitives, such as encryption schemes. OwlC, however, must link with concrete implementations. We use EverCrypt [79], a previously verified library of cryptographic primitives, via `libcrux` [30], a Rust API into EverCrypt.

Since EverCrypt is verified in F* [87], our connection from Verus to EverCrypt is trusted; fortunately, the EverCrypt API is relatively small and simple, as Figure 12 illustrates. Since EverCrypt is also verified to be constant time through type abstraction [79], we may soundly add EverCrypt's routines into SecretBuf's API in order to operate directly over SecretBufs without declassifying them.

Connecting Vest to OwlC. Owl treats message formats as opaque components of a larger protocol. During compilation, OwlC uses the Vest [27] combinators to generate a parser and serializer for every **struct** and **enum** used, and inserts calls to parsing and serialization where needed. Specifically, we use Vest's combinators for concrete bytes, integers, pairs of combinators, and tagged unions of combinators; from these, OwlC is able to synthesize verified combinators for arbitrary structs and enums written in Owl. To provide side-channel security, OwlC connects to Vest's opaque interface when parsing or serializing secrets.

8.1.2 Generated Interface

OwlC compiles each imperative procedure in the Owl source to a corresponding Rust function. Each party in Owl has a corresponding Rust **struct** whose fields contain local state (e.g., nonce counters for encryption) as well as configuration data such as pre-shared keys; in turn, the relevant Rust functions take these **struct**s in as (partially mutable) input. To use a compiled Owl protocol, larger software systems must initialize these **struct**s appropriately and call each generated function as desired.[1]

[1]Owl's type system guarantees that *any* calling order is secure.

Protecting Owl Secrets. All protocol secrets, including pre-shared and derived keys, are stored in SecretBufs; additionally, we mark generated **struct**s holding protocol state as private outside of the OwlC-generated module. Hence, by design, Rust's type system guarantees that unverified user code cannot read or modify protocol secrets. Even if the surrounding (safe) Rust code is buggy, these bugs cannot affect the security of the OwlC-verified protocols.

9 Protocol Case Studies

We demonstrate the practicality and scalability of OwlC via two large-scale case studies: formally verified, interoperable implementations of the WireGuard [36] and Hybrid Public-Key Encryption (HPKE) [12] protocols. We select these protocols since both are widely adopted in industrial and open-source applications, yet neither has a verified, high-performance implementation. In both cases, we verify and implement the core protocol in Owl, then use OwlC to extract a verified, performant Verus library; we finally use this library to build an interoperable implementation.

9.1 Protocol-level Analysis in Owl

OwlC uses Owl to prove computational security in an automated and modular fashion using information-flow types (§3). Other than new cryptographic primitives and parsing capabilities (Appendix A), OwlC does not extend Owl's core type checker; thus, we inherit the advantages and limitations of Owl for protocol-level (as opposed to implementation-level) security analysis. In particular, since Owl models corruption at the level of *keys* rather than *parties*, Owl can model KCI attacks [20], which require proving security for parties even when their static keys are compromised. However, since Owl does not model dynamic compromise [42], we cannot prove perfect forward secrecy in full generality [43]. Additionally, while Owl cannot model injective correspondences of protocol events as in CryptoVerif [21], non-injective correspondences can be modeled using refinement types; e.g., if Alice accepts a message, then Bob must have sent it.

Extensions to Owl to handle these richer security properties are orthogonal and future work. Importantly, OwlC would benefit from these improvements with little-to-no cost, since they would not affect Owl's executable fragment.

9.2 WireGuard

WireGuard is a state-of-the-art virtual private network (VPN) protocol that provides both strong security guarantees and good performance. It has seen widespread adoption, including in the Linux kernel.

WireGuard consists of two stages: the handshake, during which authentication is established and fresh transport keys are generated; and data transport, during which application data is exchanged using the derived transport keys. Each party in the protocol has a static Diffie-Hellman key, and each party also generates an ephemeral Diffie-Hellman key for each new connection. WireGuard generates transport keys from the Diffie-Hellman shared secrets using a key derivation function. During the transport phase, stateful AEAD with a counter is used to encrypt messages.

9.2.1 Verifying WireGuard in Owl

While we are not the first to formally verify cryptographic security for WireGuard [64], our proof in Owl is succinct and is attached to a verified implementation. Additionally, through Owl's type system, we give the first *modular* proof of WireGuard that does not require inlining the handshake and transport routines together.

Verification Scope. Similar to prior verification efforts [64], we do not verify the entire WireGuard executable, only the security-critical "core protocol" containing the cryptographic computations for the handshake and transport routines. We rely on unverified implementations of virtual network devices and UDP; verifying these components is left to future work.

Verification Results. Using Owl, we prove that WireGuard provides a secure channel:

- **Secrecy of transport messages:** After a successful handshake, all transport messages remain secret, unless the sender's static and ephemeral keys are compromised, or the receiver's static key is compromised.
- **Authenticity of transport messages:** In a clean session, any message decrypted by the Initiator was sent by the Responder; similarly, in a clean session, any message decrypted by the Responder was sent by the Initiator.

These properties do not cover all properties verified in prior work [64] (e.g., we do not yet verify anonymity properties, or that messages may only be received once), but are strong enough for most realistic use cases. We defer full details of our verification of WireGuard in Owl to Appendix B.

9.2.2 Building an Interoperable Implementation

Having modeled the core cryptographic routines of WireGuard in Owl as above, we use OwlC to produce a Verus library of verified implementations of those routines. Industrial WireGuard implementations typically consist of an executable with functionality beyond the core cryptographic operations, connecting to kernel APIs for networking and traffic routing. Developers would ideally write verified Verus implementations of this functionality, connecting to OwlC's library to produce a verified WireGuard executable.

To demonstrate that OwlC generates usable and performant code, we instead link our generated library to off-the-shelf industrial WireGuard implementations. We replace the handshake and transport modules in WireGuard implementations with our verified routines, but leave the other unverified components as-is, including the packet demultiplexer, timers for rekeying, and UDP network interface; none of these unverified components manipulate secret or high-integrity data. Our compilation generates simple, safe Rust interfaces for each generated method, so integrating our verified modules into pre-existing codebases is straightforward.

Our main WireGuard executable re-uses components from `wireguard-rs` [44], an officially supported, userspace implementation of WireGuard written in Rust. To demonstrate that our library can be used outside of Rust, and achieves state-of-the-art performance (§10.2), we also use FFI to connect to `wireguard-go` [37], the fastest open-source WireGuard implementation of which we are aware. In both cases, we write a small amount of glue code to allow existing code to communicate with OwlC's compiled routines.

9.3 HPKE

HPKE is a recent standardization of hybrid public-key encryption schemes, in which a Diffie-Hellman exchange is combined with a traditional symmetric cipher to allow for public-key encryption of arbitrary-length messages. HPKE has been adopted in several larger protocols, including Messaging Layer Security [11] and TLS Encrypted Client Hello [82].

HPKE uses a key encapsulation mechanism (KEM) to generate a shared symmetric secret protected by the receiver's public key; this shared secret then encrypts payload messages. It features multiple authentication modes for binding public keys to identities, plus a wide selection of primitives. Typically, HPKE exposes a *single-shot* API, in which the key encapsulation, key derivation, and encryption (or decryption) are all fused into a single procedure.

9.3.1 Verifying HPKE in Owl

Using Owl, we present the first proof of HPKE (instantiated with the Diffie-Hellman Key Encapsulation Mechanism [12]) in the plain model; that is, using realistic assumptions on hash functions, and not relying on random oracles. As with our WireGuard case study, our verification of HPKE in Owl consists of imperative procedures for each of the logical components of the HPKE sender and receiver procedures. Our model also includes HPKE's single-shot API for both sending and receiving messages.

Verification Results. We prove that HPKE (in its strongest authentication mode, AuthPSK) constructs a secure channel from the sender to the receiver; this includes both secrecy and integrity. In particular, we prove that whenever both parties are not fully compromised, messages sent using HPKE remain secret, and all received messages came from the sender.[2]

[2]The sender is fully compromised when the pre-shared key and its static and ephemeral keys are; the receiver is compromised when the pre-shared key and its static key are.

Prior formal verification efforts prove more detailed security properties for HPKE, including CCA security [4]; we leave verification of these in Owl for future work. We also leave verification of the other modes of HPKE (e.g., the Base mode, which does not use ephemeral or pre-shared keys) for future work. We defer details of our Owl verification to Appendix C.

9.3.2 Building an Interoperable Implementation

We use OwlC to produce a Verus library of the HPKE sender and receiver routines. Unlike WireGuard, HPKE is agnostic about details of message formats, network protocols, and routing, and it does not present a single uniform interface. We therefore do not need to reuse components from an off-the-shelf implementation, nor write any "glue" code to connect to one. Instead, our compiled HPKE library can be used directly in applications that rely on the protocol. We provide a simple API for the single-shot HPKE routines. Since Verus code easily links with regular Rust code, users of our library can directly call this API without depending on Verus.

10 Evaluation

We evaluate the scalability and performance of OwlC using a range of case studies. All performance tests use a Dell workstation with a 3.6GHz Intel Core i7-12700K with 32GB of RAM and running Ubuntu 20.04.

Verification Performance. As discussed in §4.2, OwlC does not emit proof hints to aid verification. We found that Verus verified our generated implementations against their ITree specifications completely automatically, even for large protocols such as WireGuard with $\sim$6,500 LoC. Furthermore, Verus verifies the code quickly, with our largest protocol (WireGuard) verifying in 13 seconds.

Figure 13 summarizes the verified protocol case studies that we examine. We discuss each group in the table in turn.

	LoC		Verif. Time (s)	
	Owl	Verus	Owl	Verus
Basic-Hash [25]	52	849	2.8	2.0
Hash-Lock [52]	62	1,489	3.8	2.9
LAK [48]	76	1,699	3.4	3.2
MW [69]	77	2,051	5.8	4.3
Feldhofer [40]	36	825	0.9	1.6
Private Auth [9]	76	1,653	10.1	3.0
Needham-Schroeder (sym) [73]	115	2,743	6.8	4.9
Needham-Schroeder (pub) [73]	95	2,299	25.8	3.5
Otway-Rees [75]	221	5,210	27.3	10.7
Yahalom (sym) [26]	175	4,028	13.4	7.4
Denning-Sacco (pub) [34]	103	2,133	7.9	3.1
Core Kerberos [57]	274	4,376	15.5	6.4
Diffie-Hellman Key Ex [35]	78	1,266	9.6	2.4
SSH Forwarding Agent [93]	223	3,130	33.0	4.5
WireGuard [36]	910	6,499	659.8	13.1
HPKE [12]	332	4,030	83.0	7.8

Figure 13: **OwlC Case Studies**. *Left to right, we show: the number of lines of Owl code; number of lines of generated Verus code; the wall-clock time for Owl to verify the protocol's security; and the wall-clock time for (multi-threaded) Verus to verify the generated code. In all cases, OwlC compiles the protocol to Verus in under a second.*

10.1 Owl Toy Protocols

The first 14 case studies shown in Figure 13 are drawn from the original Owl work [42], which used them to demonstrate Owl's utility as a protocol verifier. We successfully compile and verify implementations from each using OwlC. However, these legacy protocols are highly simplified and were not designed to be interoperable. Hence, we did not attempt to measure the performance or interoperability of OwlC's compiled code for these protocols.

10.2 WireGuard

We focus our evaluation of OwlC on the WireGuard case study (§9.2). We have two OwlC-based WireGuard executables, one using components from `wireguard-rs` [44] and one using components from `wireguard-go` [37]. We compare our OwlC-based WireGuard executables to their unverified baselines, as well as `wireguard-linux` [38], the Linux kernel WireGuard module. To our knowledge, `wireguard-linux` and `wireguard-go` are the most widely-deployed open-source WireGuard implementations; we include `wireguard-rs` since it is the only full-featured Rust implementation of which we are aware.

Interoperability. Most importantly, our implementations are able to interoperate with off-the-shelf implementations of WireGuard. We verify this by unit-testing our library in four configurations, corresponding to OwlC code or baseline code running as each of the initiator and responder of the handshake. In all four cases, both parties ultimately derive the same transport keys and are able to communicate.

Furthermore, our WireGuard implementations can present themselves using a Linux network interface; we use this to perform an end-to-end test. We use Linux network namespaces to set up a simulated network with two peers connected via a WireGuard VPN tunnel, using either the baseline code or our compiled routines as the implementation of WireGuard. In all four cases, the peers perform the handshake, derive transport keys, and communicate. Our OwlC-based WireGuard executables thus provide drop-in verified replacements for existing user-space WireGuard implementations.

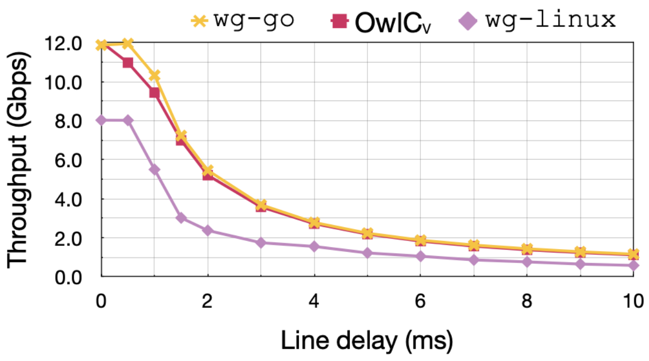

Figure 14: **End-to-End Performance Benchmark of OwlC-compiled Code**. *We report the throughput (higher is better) against the configured link delay for each configuration. Each measurement is taken for 120 seconds using the default TCP MSS (1460 B).*

10.2.1 End-to-End Transport Benchmarks

Our end-to-end benchmarks are designed to measure the performance effects on WireGuard users of using OwlC, compared to standard baselines.

We use the aforementioned virtual network topology to benchmark our code. We use `iperf3` [51], a standard network performance measurement utility, to obtain measurements of the throughput of the network. This largely measures the transport layer, since WireGuard performs its handshake only once every 120 seconds, by default.

Among the baselines, we find that `wireguard-go` and `wireguard-linux` both significantly outperform `wireguard-rs` by up to 4×, with `wireguard-go` supporting up to 50% greater throughput than the kernel module. This is unsurprising, given that `wireguard-go` is more widely deployed and heavily optimized [91] than `wireguard-rs`. We therefore focus our evaluation here on `wireguard-go`. We compare our OwlC-based code with both `wireguard-go` and `wireguard-linux`; since `wireguard-linux` does not have a single-threaded mode, our benchmarks use the default multi-threaded configuration.

For our main end-to-end benchmark, using the Linux `netem` network emulator utility [74], we configure various amounts of symmetric link latency between the two virtual WireGuard peers and measure the resulting throughput. Figure 14 shows the results. We see that the OwlC implementation performs very similarly to `wireguard-go`, incurring a maximum overhead of 6%; both OwlC and `wireguard-go` outperform the kernel module by 35%-50%. More importantly, the OwlC-compiled implementation becomes indistinguishable in performance from `wireguard-go` once we add even a tiny amount of link latency (1 ms). Considering that, for example, AWS claims "single-digit millisecond latency" even between availability zones in the same region [5], most `wireguard-go` users would notice no overhead from switching to our verified implementation; indeed, when integrated with a state-of-the-art WireGuard implementation, OwlC's WireGuard library yields an end-to-end executable that outperforms the widely-deployed Linux kernel module.

We additionally measured the performance of our code on small packets, as reported in §E.1; we observed no significant difference in performance on large packets, and a minor overhead on small packets.

We also benchmarked our executable based on `wireguard-rs`. As with our Go-based implementation, the `wireguard-rs`-based implementation using OwlC performs indistinguishably to its baseline with link latencies 2 ms or more. We observe minor overheads of up to 13% in the artificial zero-latency case, largely because `libcrux` is slower than the unverified Rust cryptographic primitives in `wireguard-rs`. Full details are reported in §E.2.

10.2.2 Micro-Benchmarks

In addition to end-to-end performance testing, we also compare the performance of our generated WireGuard handshake and transport routines in isolation. For these micro-benchmarks, we compare against `wireguard-rs` to avoid conflating overhead with the effects of comparing a language with manually managed memory (Rust) against one with garbage collection (Go). For all micro-benchmarks, we test OwlC in two configurations: using verified cryptographic primitives from EverCrypt [79] via commit `5bc3ad6` of `libcrux` [30] (abbreviated as OwlC$_V$), and using the same, unverified primitives from the baseline (abbreviated as OwlC$_B$). All data for these experiments can be found in Appendix F.

Our first micro-benchmark measures the number of handshakes per second that OwlC can perform against the baseline. When using OwlC$_V$, OwlC's implementation of WireGuard outperforms the baseline by 10%; this is mostly due to EverCrypt's more efficient elliptic curve implementation, which dominates the running time. When using OwlC$_B$, we see a negligible performance penalty of 3%.

Next, we benchmark the performance of our transport routines, measured via raw Gbps when calling the relevant functions. Here, we see a 40% performance penalty when using OwlC$_V$ when compared to the baseline, and a 15% penalty when using OwlC$_B$; this is due to the baseline containing a more efficient algorithm for encryption. Crucially, however, **this micro-benchmark is maximally pessimistic**, since we are measuring *only* the transport routines, without any notion of a network. When we measure end-to-end performance over a virtual network with 2ms of line delay (see Figure 14), the difference in throughput between OwlC and unverified implementations vanish.

		Sender	
		rust-hpke$_B$	OwlC$_V$
Receiver	rust-hpke$_B$	6,894 (—)	7,485 (+8.6%)
	OwlC$_V$	7,486 (+8.6%)	8,239 (+19.5%)

		Sender	
		rust-hpke$_B$	OwlC$_B$
Receiver	rust-hpke$_B$	6,894 (—)	7,248 (+5.1%)
	OwlC$_B$	7,259 (+5.2%)	7,640 (+10.9%)

Figure 15: **Performance Comparison of OwlC-compiled Code Against `rust-hpke`**. *Each entry indicates the number of single-shot HPKE runs per second in that configuration, along with the percentage difference from the baseline (`rust-hpke` as both sender and receiver); higher is better. The computed standard deviation is less than* 1%.

10.3 HPKE

We evaluate our HPKE case study (§9.3) by comparison to `rust-hpke` [84], a full implementation in Rust of the HPKE standard [12] which has been used in production at Cloudflare [32]. It uses a number of handwritten optimizations for performance; e.g., disallowing all allocation by default.

Interoperability. Both OwlC's HPKE code and `rust-hpke` are library crates presenting a straightforward single-shot API. To test interoperability, we wrote a small wrapper crate that generates the requisite static Diffie-Hellman key pairs and a random payload, calls either the OwlC or `rust-hpke` SingleShotSeal primitive on the payload, and finally calls either the OwlC or `rust-hpke` SingleShotOpen primitive on the resulting ciphertext. We find that in all four permutations, the decrypted message is equal to the initial payload, confirming interoperability.

Performance. To benchmark OwlC's HPKE implementation, we modify the test above to measure the combined runtime of the SingleShotSeal and SingleShotOpen routines, using `cargo bench` as we did for the WireGuard micro-benchmarks. Figure 15 shows the results of this benchmark. As with WireGuard, we report the runtime both when using verified cryptographic primitives from EverCrypt (OwlC$_V$) and using the same unverified primitives as the baseline (OwlC$_B$). We see that when using the default verified primitives, OwlC performs $\sim 20\%$ faster than `rust-hpke`; even when they use the same primitives, OwlC outperforms `rust-hpke` by $\sim 11\%$.

11 Related work

Side-channel Security for Protocols. While prior work has addressed side-channel security for cryptographic primitives [3, 23, 28, 79, 95], to our knowledge, no prior work has presented a general solution for side-channel security in cryptographic protocols. Noise* [49] also uses type abstraction to implement protocols in the Noise framework, but they use an unchecked declassification primitive; by contrast, OwlC's approach to semantic declassification guarantees that only public values are declassified. Delignat-Lavaud et al. [33] prove side-channel resistance for an implementation of QUIC, but do not develop a general-purpose solution that automatically applies to *any* protocol as OwlC does. Ironclad Apps [47] provide a scheme for declassification, but do not connect it to cryptographic constant-time.

Verified Implementations for Security Protocols. OwlC is not the first work to verify security protocol implementations. Noise* [49] automatically translates protocols from the Noise framework [77] to verified implementations in C; however, Noise* is limited to a specific class of protocols, while OwlC supports general-purpose cryptographic protocols. Moreover, relying on DY* [15], Noise* only guarantees *symbolic* security, which ignores security-critical cryptographic details; OwlC guarantees computational security, which does not make the same compromise.

Recent work [7] using Tamarin [67] translates protocol descriptions into separation logic specs, and manually proves code written in Go against those specs. A related approach [6] embeds symbolic security reasoning into a Go separation-logic verifier, enabling manual symbolic proofs on the implementations. Similarly, cv2fstar [63] translates protocols in CryptoVerif [21] to partially verified implementations in F* [87] to be completed by expert developers. In contrast, OwlC generates verified implementations fully automatically.

A line of work [17–19, 33] has formally verified implementations of high-impact security protocols, such as TLS and QUIC; however, all of these projects consist of *one-off, manual* proof efforts, wherein implementations and their specifications are hand-written in proof assistants. OwlC presents an alternative, foundational approach, based on automatic compilation to verified code.

Many prior efforts also hurt runtime performance, with examples adding $6\times$ [19] or $13\times$ [18], compared to unverified baselines. By contrast, OwlC's design facilitates optimizations that lead to state-of-the-art performance.

State Machines for Protocol Implementations. Interactive protocols can either be written *imperatively*, where network input/output are effects, or as *state machines*, using explicit handlers for messages. Prior work [7, 50] has focused on verifiably mapping between these two implementation styles. In contrast, OwlC maps Owl to Rust, which are both written imperatively, greatly simplifying verification.

Proof-Producing Compilation. While compiling into formal proofs is an old idea [70, 72, 78], our use of a deductive program verifier for automatically compiling into Rust is novel, and presents a "sweet spot" between expressivity and the automation needed for fully automatic compilation.

Tokens for Effects. While previous work uses Interaction Trees (ITrees) for specifying effects in pure proof languages [92], our work differs in regarding ITrees as abstract *permissions* to perform effects, and using these permissions by embedding them into executable code via linear typing.

Our embedding of permissions is reminiscent of prior work, such as IronSync [45]; however, IronSync uses these tokens to coordinate threads in a single program, while we use them to control the external effects of a protocol implementation.

12 Conclusion

We present OwlC, an automatic compiler from Owl protocols to efficient libraries that are verified for correctness and security. Our libraries are competitive in performance with unverified industrial implementations; we demonstrate this with two case studies, WireGuard [36] and HPKE [12], along with 14 smaller case studies from early Owl work [42].

In future, we hope to use OwlC to generate verified libraries for larger protocols, such as TLS and Signal. This will entail improvements to Owl, from which OwlC would benefit automatically, plus significant engineering effort to model the complexity of these protocols.

Ethics Considerations

Our work aims to develop formally verified, secure protocol implementations to be used by software clients in a plug-and-play manner. We do not anticipate any ethical issues arising from our work, as we are developing tools to make software more secure, and have not uncovered any vulnerabilities in other software packages, nor discovered new ways of attacking protocols or implementations.

Open Science

Our implementation of OwlC [86] is developed in a publicly available open-source repository [76], including the verified echo server example described in §7, the artifact described in §8, the large-scale protocol case studies of WireGuard and HPKE described in §9, and the 14 toy protocols from Owl [42] listed in Figure 13.

Acknowledgements

This work was supported in part by National Science Foundation (NSF) Grant No. 2224279, funding from AFRL and DARPA under Agreement FA8750-24-9-1000, and the Future Enterprise Security initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab). Pratap Singh was also funded by the NSF Graduate Research Fellowship Program under Grant No. DGE2140739.

We thank the anonymous reviewers for working with us to strengthen our paper. Michael McLoughlin provided significant help with benchmarking and profiling our case studies. Jay Bosamiya assisted with various infrastructure setup issues. Yi Cai assisted with the integration of Vest with Owl.

References

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, B. VanderSloot, E. Wustrow, S. Zanella-Béguelin, and P. Zimmermann. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[2] J. B. Almeida, M. Barbosa, and G. Barthe. Verifying constant-time implementations. In *Proceedings of the USENIX Security Symposium*, 2016.

[3] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[4] J. Alwen, B. Blanchet, E. Hauck, E. Kiltz, B. Lipp, and D. Riepel. Analysing the HPKE standard. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2021.

[5] Amazon Web Services. Availability zones. `https://docs.aws.amazon.com/whitepapers/latest/aws-fault-isolation-boundaries/availability-zones.html`, Jan. 2025.

[6] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller. A generic methodology for the modular verification of security protocol implementations. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[7] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. Basin, and P. Müller. Sound verification of security protocols: From design to interoperable implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2023.

[8] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau. An interactive prover for protocol verification in the computational model. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

[9] G. Bana and H. Comon-Lundh. A computationally complete symbolic attacker for equivalence properties. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[10] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. SoK: Computer-aided cryptography. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2021.

[11] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[12] R. Barnes, K. Bhargavan, B. Lipp, and C. A. Wood. Hybrid Public Key Encryption. RFC 9180, Feb. 2022.

[13] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte,

D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. In *Proceedings of the ACM SIGPLAN International Conference on Principles of Programming Languages*, 2019.

[14] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[15] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele. DY* : A modular symbolic verification framework for executable cryptographic protocol code. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[16] K. Bhargavan, B. Blanchet, and N. Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2017.

[17] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué. Implementing and proving the TLS 1.3 record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2017.

[18] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[19] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and S. Zanella-Béguelin. Proving the TLS handshake secure (as it is). In *Proceedings of the Annual International Cryptology Conference*, 2014.

[20] S. Blake-Wilson, D. Johnson, and A. Menezes. Key agreement protocols and their security analysis. In *Proceedings of the IMA International Conference on Cryptography and Coding*, 1997.

[21] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 2008.

[22] B. Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the Conference on Principles of Security and Trust (POST)*, 2012.

[23] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson. Vale: Verifying high-performance cryptographic assembly code. In *Proceedings of the USENIX Security Symposium*, August 2017.

[24] J. Brendel, M. Fischlin, F. Günther, and C. Janson. PRF-ODH: Relations, instantiations, and impossibility results. In *Proceedings of the Annual International Cryptology Conference*, 2017.

[25] M. Bruso, K. Chatzikokolakis, and J. Den Hartog. Formal verification of privacy for RFID systems. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2010.

[26] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, DEC Systems Research Center, Feb. 1989.

[27] Y. Cai, P. Singh, Z. Lin, J. Bosamiya, J. Gancher, M. Surbatovich, and B. Parno. Vest: Verified, secure, high-performance parsing and serialization for Rust. In *Proceedings of the USENIX Security Symposium*, 2025.

[28] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. Fact: A flexible, constant-time programming language. In *IEEE Cybersecurity Development (SecDev)*, 2017.

[29] Cloudflare, Inc. Quantifying the impact of "Cloudbleed". https://blog.cloudflare.com/quantifying-the-impact-of-cloudbleed/, Mar. 2017.

[30] Cryspen, S.A.R.L. `libcrux`: the formally verified crypto library for Rust. https://github.com/cryspen/libcrux.

[31] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[32] M. de Moura and A. Bluehs. Using HPKE to encrypt request payloads. https://blog.cloudflare.com/using-hpke-to-encrypt-request-payloads/, Feb. 2021.

[33] A. Delignat-Lavaud, C. Fournet, B. Parno, J. Protzenko, T. Ramananandro, J. Bosamiya, J. Lallemand, I. Rakotonirina, and Y. Zhou. A security model and fully verified implementation for the IETF QUIC record layer. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2021.

[34] D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[35] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22, Nov. 1976.

[36] J. A. Donenfeld. WireGuard: Next generation kernel network tunnel. https://www.wireguard.com/papers/wireguard.pdf, Jan. 2017.

[37] J. A. Donenfeld. `wireguard-go`. https://git.zx2c4.com/wireguard-go, 2023.

[38] J. A. Donenfeld. `wireguard-linux`. https://git.zx2c4.com/wireguard-linux, 2024.

[39] B. Dowling and K. G. Paterson. A cryptographic analysis of the WireGuard protocol. In *Applied Cryptography and Network Security*, 2018.

[40] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer. Strong authentication for RFID systems using the AES algorithm. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2004.

[41] A. Fromherz, A. Rastogi, N. Swamy, S. Gibson, G. Martínez, D. Merigoux, and T. Ramananandro. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2021.

[42] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno. Owl: Compositional verification of security protocols via an information-flow type system. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2023.

[43] C. G. Günther. An identity-based key-exchange protocol. In *Advances in Cryptology—EUROCRYPT'89: Workshop on the Theory and Application of Cryptographic Techniques*, 1990.

[44] M. Hall-Andersen. `wireguard-rs`. https://git.zx2c4.com/wireguard-rs, 2020.

[45] T. Hance, Y. Zhou, A. Lattuada, R. Achermann, A. Conway, R. Stutsman, G. Zellweger, C. Hawblitzel, J. Howell, and B. Parno. Sharding the state machine: Automated modular reasoning for complex concurrent systems. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2023.

[46] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2015.

[47] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2014.

[48] L. Hirschi, D. Baelde, and S. Delaune. A method for unbounded verification of privacy-type properties. *Journal of Computer Security*, 27(3), 2019.

[49] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2022.

[50] M. Ikebuchi, A. Erbsen, and A. Chlipala. Certifying derivation of state machines from coroutines. In *Proceedings of the ACM SIGPLAN International Conference on Principles of Programming Languages*, 2022.

[51] iPerf3 authors. iPerf3. https://iperf.fr/.

[52] A. Juels and S. A. Weis. Defining strong privacy for RFID. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–23, 2009.

[53] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[54] H. Kanabar, S. Vivien, O. Abrahamsson, M. O. Myreen, M. Norrish, J. Pohjola, and R. Zanetti. PureCake: A verified compiler for a lazy functional language. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2023.

[55] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, 2022.

[56] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

[57] J. Kohl and B. C. Neuman. The Kerberos Network Authentication Service (V5). RFC 1510, 1993.

[58] H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Proceedings of the Annual International Cryptology Conference*, 2010.

[59] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proceedings of the ACM SIGPLAN International Conference on Principles of Programming Languages*, 2014.

[60] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. Lorch, O. Padon, and B. Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2024.

[61] A. Lattuada, T. Hance, C. Cho, M. Brun, I. Subasinghe, Y. Zhou, J. Howell, B. Parno, and C. Hawblitzel. Verus: Verifying Rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

[62] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*, 2016.

[63] B. Lipp. *Mechanized Cryptographic Proofs of Protocols and their Link with Verified Implementations*. Theses, Université Paris sciences et lettres, June 2022.

[64] B. Lipp, B. Blanchet, and K. Bhargavan. A mechanised cryptographic proof of the WireGuard virtual private network protocol. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019.

[65] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel. A cross-protocol attack on the TLS protocol. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2012.

[66] D. A. McGrew and J. Viega. The security and performance of the Galois/counter mode of operation. In *Progress in Cryptology - INDOCRYPT*, 2004.

[67] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Proceedings of the International Confer-*

ence on Computer Aided Verification, 2013.

[68] C. Meyer and J. Schwenk. SoK: Lessons learned from SSL/TLS attacks. In *International Workshop on Information Security Applications*, 2013.

[69] D. Molnar and D. Wagner. Privacy and security in library RFID: Issues, practices, and architectures. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2004.

[70] M. O. Myreen, K. Slind, and M. J. Gordon. Extensible proof-producing compilation. In *Compiler Construction: 18th International Conference*, 2009.

[71] National Vulnerability Database. Heartbleed bug. CVE-2014-0160 http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160, Apr. 2014.

[72] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM SIGPLAN International Conference on Principles of Programming Languages*, 1997.

[73] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), 1978.

[74] netem authors. netem – network emulator. https://man7.org/linux/man-pages/man8/tc-netem.8.html.

[75] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM SIGOPS Operating Systems Review*, 21(1):8–10, 1987.

[76] OwlC code repository. https://github.com/secure-foundations/owl.

[77] T. Perrin. The Noise protocol framework. https://noiseprotocol.org/, 2021.

[78] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 1998.

[79] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Beguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2020.

[80] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the USENIX Security Symposium*, 2019.

[81] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the USENIX Security Symposium*, Aug. 2015.

[82] E. Rescorla, K. Oku, N. Sullivan, and C. A. Wood. TLS Encrypted Client Hello. Internet-Draft draft-ietf-tls-esni-18, Internet Engineering Task Force, Mar. 2024. Work in Progress.

[83] P. Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2002.

[84] M. Rosenberg. `rust-hpke`: An implementation of the HPKE hybrid encryption standard (RFC 9180). https://github.com/rozbb/rust-hpke.

[85] L. Silver, P. He, E. Cecchetti, A. K. Hirsch, and S. Zdancewic. Semantics for Noninterference with Interaction Trees. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, 2023.

[86] P. Singh, J. Gancher, and B. Parno. OwlC archival code repository. https://doi.org/10.5281/zenodo.15605318.

[87] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM SIGPLAN International Conference on Principles of Programming Languages*, 2016.

[88] N. Swamy, T. Ramananandro, A. Rastogi, I. Spiridonova, H. Ni, D. Malloy, J. Vazquez, M. Tang, O. Cardona, and A. Gupta. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022.

[89] The Rust Project Developers. cargo-bench(1)—The Cargo Book. https://doc.rust-lang.org/cargo/commands/cargo-bench.html.

[90] T. Wallez, J. Protzenko, and K. Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[91] J. Whited and J. Tucker. Userspace isn't slow, some kernel interfaces are! https://tailscale.com/blog/throughput-improvements, Dec. 2022.

[92] L.-y. Xia, Y. Zakowski, P. He, C.-K. Hur, G. Malecha, B. C. Pierce, and S. Zdancewic. Interaction trees: Representing recursive and impure programs in Coq. In *Proceedings of the ACM SIGPLAN International Conference on Principles of Programming Languages*, 2019.

[93] T. Ylonen. The secure shell (SSH) transport layer protocol. RFC 4253, 2006.

[94] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, 2021.

[95] J. K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

A Modifications to the Owl Type System

Here, we describe additions we made to Owl's type checker in order to support realistic protocols, including WireGuard and HPKE.

Plain-Model KDFs. First, to support sophisticated key exchange protocols such as WireGuard, OwlC extends Owl's support for Key Derivation Functions (KDFs), which are used to convert shared secrets into uniformly random keys. In particular, the extended Owl type system now supports *plain-model KDFs*, which do not require assuming that the protocol has access to an idealized random oracle.

In particular, OwlC extends Owl with an abstraction of the HKDF [58] primitive, which combines the Extract and Expand steps into one function with three arguments: the *salt*, which may be optionally filled with secret uniform randomness; the *input keying material*, or IKM, which may contain other forms of shared secrets (e.g., Diffie-Hellman shared secrets); and a *context*, which is used to bind the output key to public data (e.g., algorithm parameters).

To verify security protocols, we assume that HKDF satisfies three (standard) assumptions: that HKDF is a pseudo-random function (PRF) when the salt is a uniformly random secret key; similarly, HKDF is a PRF when the IKM is a uniform secret key; and a mild variant of PRF-ODH [24], which states that HKDF is also a PRF when the IKM contains a Diffie-Hellman secret.[3]

The typing rules for KDFs in Owl are similar in spirit to the original ones for Random Oracles: when the KDF is fed with appropriate secrets, the KDF outputs a value of type Name(KDF⟨pos⟩(salt, ikm, ctx)); this states that the value is a fresh cryptographic secret, where pos contains metadata about how many bytes were extracted for the KDF, and salt, ikm, and ctx are the corresponding inputs to the KDF.

Stateful AEAD. Additionally, OwlC extends Owl to support the most common form of symmetric encryption today, *authenticated encryption with associated data*, or AEAD [83]. AEAD is quite similar to the basic form of symmetric encryption already supported by Owl, but it has two additional features useful for real-world protocols: *counter-driven nonces* and *additional associated data*, or AAD, which can authenticate data outside of the secret message.

Real-world AEAD schemes (e.g., AES-GCM [66]) allow the developer to choose a nonce, so long as no nonce is used twice for encrypting. The near-universal choice for the nonce is to use a counter that is stored with the encryption key and incremented each time it is used.

To support this in Owl, OwlC includes a new *name type* for Stateful AEAD (st_aead), which embeds a monotonic counter into the specification for the encryption key; thus, by construction, no nonce can be reused. Additionally, for more sophisticated uses, OwlC allows this counter to be mapped under a user-chosen *bijection*; for example, our HPKE case study uses nonces of the form $b \oplus n$, where n is the counter, and b is a random *base nonce* that obscures network traffic.

Byte-Precise Data Types. Finally, we extend Owl's type system to incorporate low-level format details necessary for interoperability. For example, the WireGuard network format includes magic constants to differentiate various messages; to support this, we include *singleton types* Const(N) in Owl structs, which hold exactly the magic constant N.

[3]For WireGuard, we rely on PRF-ODH as-is; but for HPKE, we need to slightly extend PRF-ODH to handle IKM values of the form $g^{wx}||g^{yz}$.

B Verifying WireGuard in Owl

Following Owl's design philosophy, we write the procedures for the Initiator and Responder in Owl as imperative programs, closely resembling executable implementations. By using Owl's support for *indexed* code, we prove security in a setting with an arbitrary number of Initiators and Responders who participate in arbitrarily many sessions.

By using Owl, we guarantee information-flow security by default. In particular, *any* secret not corrupted by the adversary cannot flow to the network. The only valid network messages must have the information-flow label adv, which stands for fully public data.

For data transport, we model secrecy using abstract labels, [channel_secret_init_to_resp⟨@n,m⟩] and [channel_secret_resp_to_init⟨@n,m⟩], for the two unidirectional channels between the nth Initiator and the mth Responder. Via information flow, we prove that the Initiator's channel to the Responder is compromised only when both its static and ephemeral keys are compromised, or when the Responder's static key is compromised. We prove the equivalent result for the Responder's channel to the Initiator.

Handshake Security via Clean Sessions. The central challenge of verifying WireGuard's handshake is proving security in the presence of *unauthenticated key shares*, since ephemeral public keys are passed in plaintext without any form of authentication. To specify security in this setting, we must, as in prior work [39, 64], define a *clean session*, which defines the conditions under which the secrecy and integrity of derived keys are guaranteed.

At the end of the handshake, the Initiator derives two unidirectional transport keys. The Owl type for these keys is, roughly, **if** init_clean then SecretName(...) **else** Data⟨adv⟩. Hence, if the initiator's session is clean, the keys derived are both secret and authentic. If the session is not clean, the keys are arbitrary, possibly corrupted, public data.

On the Initiator side, a session is clean when it is invoked with a secret pre-shared key (if it is using a pre-shared key), or at least one of the four Diffie-Hellman computations (Initiator's ephemeral/static key $\leftrightarrow$ Responder's ephemeral/static key) involves two keys that are uncompromised (i.e., the keys are actual secrets). Since the Initiator may be invoked with a fake ephemeral key for the Responder, an "uncompromised" ephemeral key for the Initiator also requires that the ephemeral key corresponds to the intended Responder. The Responder's notion of clean sessions is similar, except that the Responder's cleanliness predicate also requires that, in the absence of a valid, secret pre-shared key, the received ephemeral key is a valid ephemeral key of some Responder.

Key Confirmation for the Responder. An interesting quirk of WireGuard is that security for the Responder is only guaranteed after it receives a transport message from the Initiator, which confirms that the Initiator has completed the handshake. In prior proof efforts, both formally verified and otherwise [36, 64], this restriction breaks modularity between the two phases, resulting in a proof that combines part of the transport phase with the handshake phase.

Notably, our security proof in Owl does not require breaking modularity in this way. Instead of combining the phases, we use Owl's support for dependent types to encode a form of *typestate*. The Responder's derived keys are contained in a struct with a field received : Bool, which indicates whether the first transport message has been received. In turn, the transport keys have a type of the form

$$\begin{cases} \mathsf{SecName}(..) & \textbf{if}\ \mathsf{resp\_clean}\ \&\&\ (\mathsf{received}\ \|\ \mathsf{correct\_key}) \\ \mathsf{JunkSecret}(..) & \textbf{if}\ \mathsf{resp\_clean}\ \&\&\ !\mathsf{received}\ \&\&\ !\mathsf{correct\_key} \\ \mathsf{Data}\langle\mathsf{adv}\rangle & \text{otherwise} \end{cases}$$

Above, correct_key indicates whether the adversary chose to deliver the correct keys in the handshake, SecName(..) is the correct, secret key, and JunkSecret(..) is a secret but useless encryption key that is specified to not encrypt any plaintexts. Without receiving a message from the transport, the Responder cannot guarantee that it is using the correct key instead of the junk secret; however, after receiving the key, in a clean session, the Responder is guaranteed to be using the authentic transport key. By using the above type signature for transport keys in Owl, we can separately typecheck the handshake and transport phases of WireGuard.

C Verifying HPKE in Owl

Our Owl proof of HPKE implements the strongest authentication mode (AuthPSK), in which the key for the symmetric cipher is derived using a pre-shared key and two Diffie-Hellman computations—one between the sender and receiver's public keys, and one between the sender's ephemeral key and the receiver's public keys. Our Owl implementation focuses on AuthPSK for its maximal security properties, but it can readily be adapted to handle the other modes as well.

Security Assumptions for AuthPSK. Interestingly, in the plain model (i.e., without a random oracle), AuthPSK requires a cryptographic assumption that, to our knowledge, has not yet been defined in the literature. To derive a shared symmetric secret, the receiver computes (roughly) a hash of the form $H = \mathsf{hkdf}(\dots, \text{“HPKE\_v1”} \mathbin{+\!\!+} g^{s_S s_R} \mathbin{+\!\!+} h^{s_R}, \dots)$ where hkdf is the HKDF key derivation function [58], s_S and s_R are the sender and receiver's static Diffie-Hellman keys, and h is an unauthenticated group element that is meant to be interpreted as the sender's public ephemeral key. In the random oracle model, the secrecy of the hash H follows from the Gap Diffie-Hellman assumption [24]; however, in the plain model, one must use a more precise assumption.

The state of the art in the plain model is PRF-ODH [24], which states that for Diffie-Hellman secrets x and y, hashes of the form $\mathsf{hkdf}(\dots, h^x, \dots)$ are secret whenever $h = g^y$, even when the adversary may obtain hashes using arbitrary other group elements $h \neq g^y$.

While useful for cases such as TLS [24], PRF-ODH is not applicable for HPKE, since the second input to hkdf (the *input keying material*, or IKM) has more structure. Thus, for HPKE, we assume a generalization of PRF-ODH wherein the adversary may construct arbitrary contexts for the IKM of the form $\cdot \mathbin{+\!\!+} h^x \mathbin{+\!\!+} \cdot$. Since the pseudorandomness of HKDF hinges on the min-entropy of the IKM [58], adding extra information to the IKM cannot harm security.

D Compiler Architecture and Passes

As described in §3, OwlC's compilation pipeline consists of two compilers: a simple, trusted *specification compiler* that translates the Owl program into an equivalent ITree; and an untrusted *implementation compiler* that generates the executable Rust functions and Verus spec/proof annotations to prove adherence to the ITree specification.

Concretization and Spec Compilation. Before compilation, OwlC performs *concretization*, which erases ghost code from the Owl program. Additionally, this phase collapses the fine-grained information-flow lattice of the Owl program to just public and secret: public values are those whose information-flow label is public under all corruption scenarios, while secret values are those whose information-flow label may not be public in at least one corruption scenario. The output of concretization corresponds to the example protocol in Figure 2.

Given the concretized protocol, OwlC performs *spec compilation* to produce the ITree representation of each protocol routine. While this phase of the compiler is trusted for correctness, the translation from concretized Owl to Interaction Trees is a simple syntactic transformation, implemented as a single traversal of the AST.

As described in §6, however, the implementation compiler must keep track of type information, both to generate valid Rust code and to enable optimizations. OwlC does so over three passes.

Format Type Elaboration. First, the compiler annotates every node in the concretized AST with a *format type*. In essence, format types encode the concrete representation of abstract terms in Owl, without committing to Rust-level implementation details. Format types are defined by the following grammar:

$$\tau \ ::= \ \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{usize} \mid \mathsf{ADT}(s) \mid \mathsf{Option}\langle\tau\rangle \mid \mathsf{buffer}\langle\ell\rangle[N?] \mid \mathsf{hexconst}(v)$$

Here, $\mathsf{ADT}(s)$ corresponds to Owl **struct**s and **enum**s, which are represented in OwlC's output as Rust **struct**s and **enum**s. $\mathsf{buffer}\langle\ell\rangle[N?]$ encompasses byte buffers: ℓ represents the information-flow secrecy of the buffer (either public or secret); $N?$ represents the length of the buffer, and may be present for buffers of a known length (e.g., a key), or absent for buffers of unknown lengths (e.g., network messages). We additionally have $\mathsf{hexconst}(v)$ for byte constants, such as those appearing in network formats, which are kept separate from the buffer⟨l⟩[N?] type to facilitate the generation of Vest parsers and serializers for those formats.

Lowering to Rust Types. Next, the compiler translates the elaborated AST into a new AST annotated with Rust types. The full grammar of Rust types that OwlC uses is shown below:

$$\tau \ ::= \ \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{usize} \mid \mathsf{Vec}\langle\mathsf{u8}\rangle \mid \&[\mathsf{u8}] \mid [\mathsf{u8};\mathsf{N}] \mid \mathsf{ADT}(s) \mid \mathsf{Option}\langle\tau\rangle \mid \mathsf{PublicBuf} \mid \mathsf{SecretBuf}$$

Compared to the format types, we have eliminated $\mathsf{hexconst}(v)$, and replaced $\mathsf{buffer}\langle\ell\rangle[N?]$ with our OwlC-specific PublicBuf and SecretBuf, as well as the standard Rust buffer types of Vec⟨u8⟩, &[u8], and [u8;N].

This pass performs a limited amount of analysis to determine the right Rust-level representation of each intermediate value, and it inserts explicit Rust-level conversion operations (such as Vec::as_slice()) as necessary. Importantly, the compiler only has to emit a small set of basic operations, corresponding to cryptography, parsing/serialization, network interaction, and other basic operations such as bytestring concatenation. Thus, OwlC simply hardcodes the type signatures of the corresponding Rust functions and uses these signatures to inform the lowering pass.

Lowering is also responsible for generating the fused buffer operations described in §6. To do so, the lowering pass tracks whether each intermediate value can be computed by a fused operation; it then inserts the fused operation whenever that value is used. To avoid redoing expensive computations, this pass also tracks whether the result of a fused operation is used multiple times, in which case we fall back to allocating a buffer for the intermediate value.

Code Generation. Finally, the compiler pretty prints the lowered AST as Verus concrete syntax. With the AST already annotated with Rust types, code generation can be implemented as a simple traversal of the AST, without needing to track any contextual information. Code generation also emits Rust **struct** and **enum** definitions, and Vest combinator definitions, for the data type definitions in the Owl program.

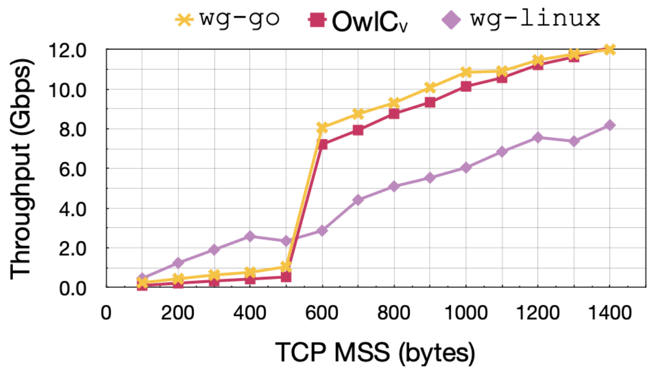

Figure 16: **End-to-End Performance Comparison Against Packet Payload Size**. *We report the throughput (higher is better) against the TCP MSS. Each measurement was taken for 120 seconds, with zero link latency.*

E Additional End-to-End Benchmarks

E.1 Comparison to `wireguard-go`

WireGuard Packet Size Benchmark. In addition to our link-latency benchmark (§10.2), we measure the performance of OwlC's WireGuard implementation on small packets. We do so by adjusting the TCP maximum segment size, which controls the size of the WireGuard packet payloads. We measure performance in the worst-case scenario where there is no latency between the two parties.

Figure 16 shows the results. Comparing OwlC to `wireguard-go` and `wireguard-linux`, we observe broadly similar performance between OwlC and `wireguard-go`. Note that `wireguard-go` is optimized for large packets, explaining the significant throughput jump at MSS 600; since OwlC uses the same network-handling code as `wireguard-go`, OwlC inherits the same performance behavior.

E.2 Comparison to `wireguard-rs`

We repeat the benchmarks described in §10.2.1 and §E.1, but using our OwlC-based WireGuard executable using components from `wireguard-rs`. This implementation suffers from significantly poorer performance than `wireguard-go`, or our OwlC implementation based on `wireguard-go`, since `wireguard-go` features significant performance optimizations. However, our `wireguard-rs`-based implementation is more similar to the intended use case of OwlC, featuring application Rust code calling directly into our WireGuard library, rather than an FFI bridge to Go code.

Our `wireguard-rs` benchmarks run both OwlC and `wireguard-rs` in single-threaded mode. Similar to our microbenchmarks, we report performance for OwlC in two configurations: OwlC$_V$, using `libcrux` for cryptographic primitives; and OwlC$_B$, using the unverified pure-Rust primitives used in `wireguard-rs`. Our OwlC$_B$ results isolate the specific

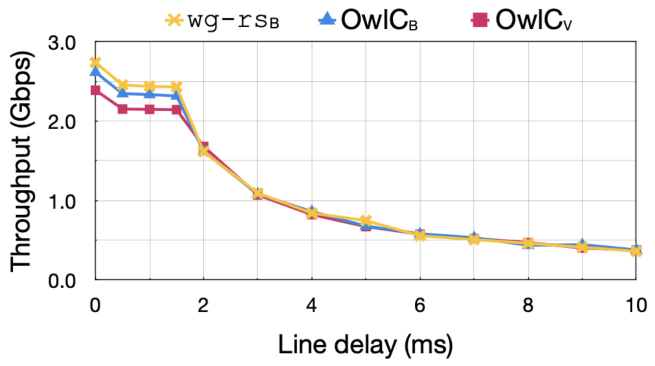

Figure 17: **End-to-End Performance Comparison of OwlC-compiled Code Against `wireguard-rs`**. *We report the throughput (higher is better) against the configured link delay for each configuration. Each measurement is taken for 120 seconds using the default TCP MSS (1460 B).*

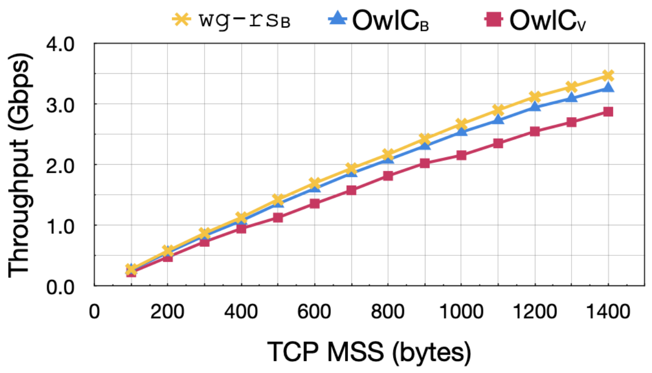

Figure 18: **End-to-End Performance Comparison Against Packet Payload Size For OwlC-compiled Code Against `wireguard-rs`**. *We report the throughput (higher is better) against the TCP MSS. Each measurement was taken for 120 seconds, with zero link latency.*

		Initiator	
		`wg-rs`$_B$	OwlC$_V$
Responder	`wg-rs`$_B$	4,914 (—)	5,102 (+3.8%)
	OwlC$_V$	5,162 (+5.0%)	5,392 (+9.7%)

		Initiator	
		`wg-rs`$_B$	OwlC$_B$
Responder	`wg-rs`$_B$	4,914 (—)	4,809 (−2.1%)
	OwlC$_B$	4,873 (−0.8%)	4,759 (−3.1%)

Figure 19: **Performance Comparison of OwlC-compiled Code Against `wireguard-rs` for WireGuard Handshake**. *Each entry indicates the number of handshakes/second in that configuration, along with the percentage difference from the baseline (`wireguard-rs` as both initiator and responder); higher is better. In all measurements, the computed standard deviation is less than 2%.*

impact of our work, ignoring the effect of using a different cryptographic library.

WireGuard Line Delay Benchmark. We use the same virtual network setup as described in §E.2 to configure symmetric link latency and measure throughput. Figure 17 shows the results. We see that at zero link latency, we observe a performance penalty of approximately 13% for OwlC$_V$, whereas with OwlC$_B$, the performance overhead drops to only 4%. This indicates that code compiled by OwlC adds only 4% overhead and that the baseline's unverified implementation of ChaCha20-Poly1305 (drawn from OpenSSL) outperforms the verified implementation in EverCrypt. Again, we note that with a link latency of just 2 ms, OwlC performs indistinguishably from the baseline.

WireGuard Packet Size Benchmark. We additionally compare OwlC's performance to `wireguard-rs` on small packets, using the artificial worst-case of zero link latency. Figure 18 shows the results. We observe a consistent throughput penalty of 13-21% for OwlC$_V$. Switching to the same primitives as the baseline lowers the overhead to only 3-5%.

F Details on Micro-Benchmarks

Our micro-benchmarks run specific functions from the OwlC and baseline code bases; during normal execution these functions would be just one component among many in the implementation. We also remove any communication cost by running the relevant routines from two separate WireGuard devices in the same address space, so that "reading" and "writing" packets can be done by just reading and writing to a buffer in memory. We measure the runtime of these benchmarks using Rust's standard `cargo bench` infrastructure [89], which runs at least 300 iterations of the benchmark and continues to run until the overall median converges to a stable value, reporting the median and scaled median absolute deviation of the runtimes.

Handshake. Our handshake micro-benchmark initializes two WireGuard devices and performs one handshake between them, checking that both devices derive the same transport keys. We test this benchmark with both OwlC and `wireguard-rs` code for both devices. Figure 19 shows the results, for both OwlC$_V$ (using verified cryptographic primitives from EverCrypt [79]) and OwlC$_B$ (using the same unverified pure-Rust primitives as `wireguard-rs`).

Transport. Our transport micro-benchmark isolates just the routines that generate and handle WireGuard packets: it repeatedly generates a random payload, calls the WireGuard packet creation routine on that payload, then calls the WireGuard packet processing routine on the output of packet creation. It omits all state management, routing, network communication, and packetization/stream reconstruction functionality; it is thus an artificial worst-case measurement of our

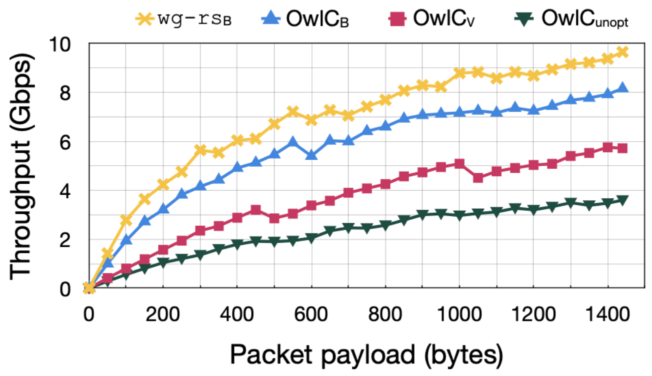

Figure 20: **Performance Comparison of OwlC-compiled Code Against `wireguard-rs` for WireGuard Transport**. *Effective line delay is zero (see Figure 14). We report the effective throughput (higher is better) against the packet payload for each configuration. This micro-benchmark isolates just the routines for sending and receiving packets.*

code, since all of that functionality is required for a working WireGuard implementation.

Figure 20 shows the results in terms of the effective implied throughput of the packet-sending and packet-receiving routines at a range of packet sizes. The default configuration of OwlC incurs an overhead of about 40% at the largest packet size. This is due in large part to the faster implementation of ChaCha20-Poly1305 in the baseline; when using that implementation, OwlC incurs an overhead of about 15%. We additionally report performance for $\text{OwlC}_{\text{unopt}}$, a version of our WireGuard implementation compiled without our optimizations (including those in §6). Despite using the faster unverified ChaCha20-Poly1305 implementation, $\text{OwlC}_{\text{unopt}}$ incurs a significant overhead of up to 80%, underscoring the importance of our optimizations. We leave the task of improving performance with verified cryptography to future work.