Vest: Verified, Secure, High-Performance Parsing and Serialization for Rust

Yi Cai[†][*] Pratap Singh[‡] Zhengyao Lin[‡] Jay Bosamiya[¶][*]
Joshua Gancher[§][*] Milijana Surbatovich[†] Bryan Parno[‡]

[¶]*Microsoft Research* [§]*Northeastern University* [‡]*Carnegie Mellon University*
[†]*University of Maryland, College Park*

Abstract

Many software vulnerabilities lurk in parsers and serializers, due to their need to be both high-performance and conformant with complex binary formats. To categorically eliminate these vulnerabilities, prior efforts have sought to deliver provable guarantees for parsing and serialization. Unfortunately, security, performance, and usability issues with these efforts mean that unverified parsers and serializers remain the status quo.

Hence, we present Vest, the first framework for high-performance, formally verified binary parsers and serializers that combines expressivity and ease of use with state-of-the-art correctness and security guarantees, including—for the first time—resistance to basic digital side-channel attacks. Most developers interact with Vest by defining their binary format in an expressive, RFC-like DSL. Vest then generates and automatically verifies high-performance parser and serializer implementations in Rust. This process relies on an extensible library of verified parser/serializer combinators we have developed, and that expert developers can use directly.

We evaluate Vest via three case studies: the Bitcoin block format, TLS 1.3 handshake messages, and the WebAssembly binary format. We show that Vest has executable performance on-par (or better) than hand-written, unverified parsers and serializers, and has *orders of magnitude* better verification performance relative to comparable prior work.

1 Introduction

Code that handles untrustworthy binary data, e.g., to process a network packet or decode a file, is often buggy and hence exploitable [26, 27, 44, 46, 61]. This is primarily due to the conflicting goals of high performance and adherence to existing, complex binary formats. Trying to meet both goals drives developers to manually write parsing libraries in unsafe, low-level languages, such as C. Making matters worse, these libraries must also handle serialization—often with devastating security consequences if parsing and serialization are not mutual inverses.

In response, the research community has proposed a number of *formally verified* frameworks to produce parsers and serializers that guarantee correctness and security [25, 50, 54, 57, 59]. However, these tools are not yet mature enough for software engineering at large: they can be slow to execute [25, 59], slow to verify [50, 54] (making it painful to iterate on new formats), missing functionality [54] (verifying parsers but not serializers), lacking guarantees about side-channel resistance [25, 50, 54, 59], or exposing difficult, developer-unfriendly low-level APIs [50].

Our Approach. We present Vest, the first framework for formally verified parsing/serialization that combines expressivity and ease of use with state-of-the-art correctness and security guarantees, including—for the first time—resistance to basic digital side-channel attacks (e.g., through timing). Given a description of a binary format in our simple, RFC-like DSL (VestDSL), Vest generates high-performance parser and serializer implementations in safe Rust [35, 43] that are carefully tailored to ensure automatic verification by Verus [37, 38], a deductive program verifier for Rust code. In turn, the developer may use this generated code in ordinary Rust projects through intuitive, simple interfaces and easy `cargo` integration.

VestDSL is expressive enough to capture real-world formats. A key novelty of VestDSL is its robust support for both *variant* and *dependent* formats. Variant formats include, e.g., optional fields, as well as unions that are explicitly tagged, implicitly tagged, or even untagged. VestDSL's dependent formats support dependencies both *within* formats (e.g., for tag-length-value formats), and *external* to formats (crucial for formats that depend, say, on a protocol-level state machine).

For every VestDSL format, Vest proves that the corresponding parsers and serializers are: *safe* (ruling out buffer overflows, integer overflows, use-after-free, ...), *correct* (parsers and serializers are mutual inverses, ruling out format confusion [59] and malleability [50] attacks), and *side-channel free* (specifically, source-level freedom from secret-dependent memory accesses or timing). In short, Vest relieves developers from worrying about trade-offs between security, correctness, and ease of development.

[*]Work done in part while at Carnegie Mellon University.

The above guarantees are driven by VestLib, a library of formally verified, high-performance parser and serializer *combinators* we have developed. A parser combinator [28] is a function that takes in one or more parameters (some of which may be parser combinators) and returns a new parser (and likewise for serializer combinators). While VestDSL serves as the primary workflow for most developers, each VestLib combinator has a consistent interface for their specification, proof, and implementation. This uniform design enables experts to straightforwardly extend VestLib with custom combinators tailored for complex formats. Indeed, recent projects have already leveraged VestLib extensively—for instance, to implement provably secure parsers and serializers for X.509 certificate formats [41] and to automatically construct side-channel resistant parsers for security protocols [52].

Evaluation. We evaluate Vest on three real-world case studies: the Bitcoin block format, TLS 1.3 handshake messages, and WebAssembly binaries, all encoded in VestDSL.[1] Due to our careful design choices and Verus's powerful proof automation, each case study's generated implementations and security proofs verify in seconds (compared to hours for previous work [50, 54]), making it feasible to include parser/serializer verification in CI pipelines. Thanks to our highly-efficient parser and serializer implementations, our generated code's performance is competitive with (or faster than) state-of-the-art, hand-optimized Rust implementations for all three of our case studies.

Limitations. Vest's guarantees depend on the correctness of our verification tool, Verus, as well as the correctness of the Rust compiler (`rustc`). In particular, Vest provides side-channel resistance guarantees only at the level of Rust source programs; compilation with `rustc` may undermine these guarantees. Prior work [15, 23] has developed constant-time preserving compilers; applying these ideas to Rust is future work.

Contributions. In summary, we present:

1. The design of VestDSL, which combines readability with real-world expressivity and the ability to automatically produce verified parsers and serializers without requiring any verification expertise from the developer.
2. An elegant, trait-based treatment of parser and serializer combinators in Rust, supporting readable code and automated proofs.
3. The first approach to automatically and generically produce parsers and serializers that guarantee freedom from basic digital side channels.
4. Vest, a concrete instantiation of these techniques, and the first verified parser/serializer framework in Rust.
5. A detailed evaluation on three real-world case studies.

2 A Developer's View of Vest

Vest is a toolchain for formally defining binary formats and generating verified, efficient parsers and serializers from those definitions. Figure 1 illustrates Vest's workflow. First, the programmer writes a formal definition of their format (possibly derived from an existing external format specification) in VestDSL, a declarative, RFC-like domain-specific language for defining binary formats. Next, the VestDSL compiler type-checks the definition and generates a Rust module. This module contains four key components:

1. **Data Type Definitions:** Rust data types that mirror the structure of the format.
2. **Trusted Specifications:** A set of parser and serializer specifications (written in Verus [37, 38]) composed of VestLib combinators that represent the format's parsing and serialization behaviors.
3. **Correctness and Security Proofs:** *Machine-checkable* proofs of correctness and security for the specifications, also written in Verus and constructed from existing combinator proofs in VestLib.
4. **Efficient Implementations:** *Formally verified*, *efficient*, *safe* Rust implementations of the parser and serializer for the format, again composed of VestLib combinators.

Once the generated module is verified by Verus, the programmer can integrate it via `cargo` into larger verified projects [41, 52], or into unverified Rust projects. The latter can use it without depending on Verus.

Trusted vs. Untrusted Components. For *security*, Vest maintains a minimal trusted computing base (TCB) that consists only of Verus/Rust and the portion of VestLib that defines the formal properties of VestLib's specification and implementation combinators ($<$ 150 LoC). As illustrated in Figure 1, users need *not* trust the majority of VestLib, the entire VestDSL compiler, nor the generated Rust module for security, since it (or its output) is mechanically verified by Verus.

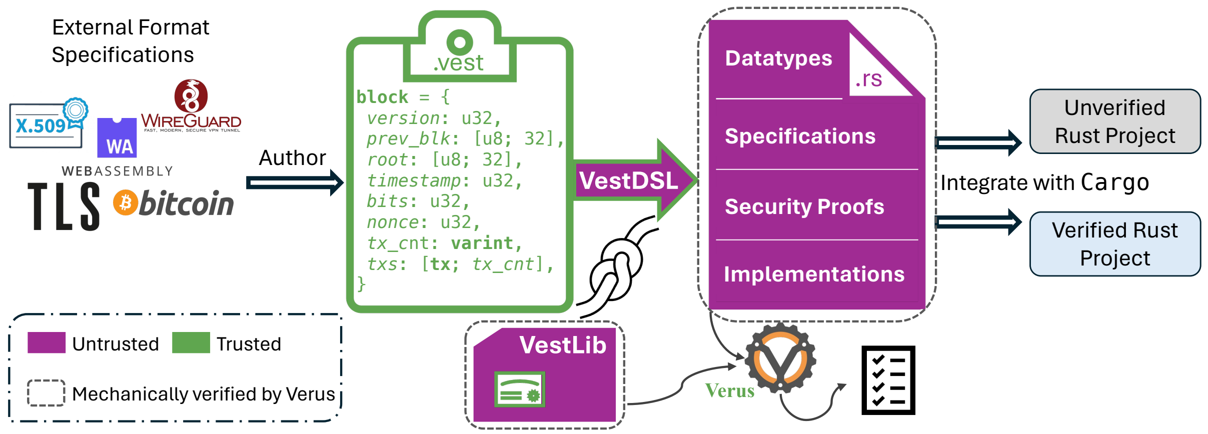
For end-to-end *correctness* of the generated implementation, however, we additionally trust the developer-authored format definition in VestDSL, as well as the portion of the compiler that lowers the DSL definition to the VestLib spec combinators. Crucially, the implementation generation process remains *untrusted*, as the generated implementations are verified to match the generated specs.

Next, we describe Vest's key features at a high-level and explain how VestDSL and VestLib fit together.

2.1 VestDSL: Formal Definitions of Binary Formats

We start by illustrating the challenges of defining binary formats with existing informal approaches, and how VestDSL

[1] Vest and our three case studies are publicly available from https://github.com/secure-foundations/vest.



```
block = {
  version: u32,
  prev_blk: [u8; 32],
  root: [u8; 32],
  timestamp: u32,
  bits: u32,
  nonce: u32,
  tx_cnt: varint,
  txs: [tx; tx_cnt],
}
```

Figure 1: **Vest Workflow**. *For security, we trust VestLib's formal specs and Verus, but everything else is untrusted, since it (or its output) is mechanically verified. For compatibility with the intended format, we also trust the developer-authored format definition in VestDSL, and the portion of the compiler that lowers the DSL definition to the VestLib spec combinators.*

addresses this challenge by providing a formal, unambiguous way to define binary formats.

Defining Binary Formats, Informally. Existing binary formats are typically defined in RFCs or other documents, which often mix English prose with C-like type definitions. This informal approach can contain implicit constraints and ambiguities; e.g., TLS 1.3 defines the `HelloRetryRequest` message as:

> *"For reasons of backward compatibility with middleboxes, the HelloRetryRequest message uses the same structure as the ServerHello, but with Random set to the special value of the SHA-256 of HelloRetryRequest: CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91 C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C "* [51]

Instead of explicitly defining the `HelloRetryRequest` message, the RFC *warns* the implementor that upon receiving a handshake message with type `server_hello`, the parser should first check if the `Random` field is equal to the special byte sequence above. If so, the parser should treat the following bytes as a `HelloRetryRequest` message, which has the same *structure* as a `ServerHello` message but very different *behavior* in the context of the handshake protocol.

As another example, consider the TLS ClientHello padding extension, whose format descriptions and constraints are defined "by example":

> *"The "extension_data" for the extension consists of an arbitrary number of zero bytes. For example, the smallest "padding" extension is four bytes long and is encoded as 0x00 0x15 0x00 0x00. The client MUST fill the padding extension completely with zero bytes, although the padding extension_data field may be empty."* [36]

Even existing frameworks for verified binary formats lack the expressivity to properly capture such ad-hoc constraints for conditional or dependent formats [50, 54, 59]. Instead they rely on *staged parsing* or require the programmer to manually implement and verify the ad-hoc constraints.

Defining Binary Formats, Formally. VestDSL is an expressive, declarative, unambiguous format description language that comes with a set of built-in constructs for *formally* defining binary formats. The programmer can use VestDSL to express primitive formats (e.g., fixed-/variable-size integers, byte arrays), formats with restrictions (e.g., constants, integers within a range), structured formats (e.g., sequencing and choice of formats), or even inter- and intra-dependencies between formats (e.g., parametric formats like tagged unions).

Each of the high-level DSL constructs has a precise *formal semantics* in the form of an equivalent VestLib combinator. As a simple example, VestDSL's structure format `a = { x: u8, y: u8 }` specifies a structure with two concatenated fields x and y that are both 8-bit unsigned integers, without any padding. The formal semantics is a parser that reads two bytes in sequence from the input buffer and maps them to the fields x and y of the struct, and a serializer that accesses the fields x and y of the struct and writes them to the output buffer in sequence.

VestDSL enables the programmer to capture the subtleties of binary formats in a precise, unambiguous manner, in contrast to informal prose descriptions. For example, in Figure 2, we show how a programmer can formally define the choice between `ServerHello` and `HelloRetryRequest` messages, as well as the dependencies and constraints in the ClientHello padding extension in VestDSL. First, we define a format named `sh_or_hrr` that includes a *dependent field* @random and a field payload that *chooses* between the two sub-formats based on the value of @random. The `choose` construct functions like the `match` construct found in Rust or similar languages; it takes in a dependent field (denoted with the @ symbol) and pattern-matches on the field's value to select between dif-

```
sh_or_hrr = {
  // some fields elided
  @random: [u8; 32],
  payload: choose (@random) {
    [0xcf, 0x21, 0xad, 0x74, 0xe5, 0x9a, 0x61, 0x11,
      0xbe, 0x1d, 0x8c, 0x02, 0x1e, 0x65, 0xb8, 0x91,
      0xc2, 0xa2, 0x11, 0x16, 0x7a, 0xbb, 0x8c, 0x5e,
      0x07, 0x9e, 0x09, 0xe2, 0xc8, 0xa8, 0x33, 0x9c] ⇒
        hello_retry_request,
    _ ⇒ server_hello,
  },
}
```

```
padding_extension(@len: u16) = {
  extension_data: [u8; @len] ≫= Vec<zero_byte>,
}
```

Figure 2: **VestDSL Examples**.

ferent formats. In this case, if the value of the field @random is the special byte sequence defined in the RFC, the format is a `hello_retry_request` message; otherwise (denoted by the wildcard _), it is a `server_hello` message, where each message adheres to a precisely defined format. Similarly, we formally define the `padding_extension` format. This format is parameterized by a `u16` value that specifies the length of the extension_data field, which is constrained to be a vector of zero bytes (denoted by ≫= `Vec<zero_byte>`).

2.2 VestLib: Verified Parsers & Serializers

Once the programmer has formally defined their binary format in VestDSL, the VestDSL compiler generates the corresponding Rust data type definitions, efficient parser and serializer implementations, and, for proof purposes, a set of annotations in Verus [37, 38], a state-of-the-art, semi-automated program verifier for Rust.

Background on Verus. Although Vest can be directly integrated into unverified Rust projects via `cargo`, understanding some of Verus's key features is valuable, especially since part of the generated code is written in Verus (Figure 1). Specifically, Verus enhances each Rust function by allowing annotations for pre-conditions (`requires`) and post-conditions (`ensures`). These annotations are automatically translated into logical formulas and verified by the underlying SMT solver. Moreover, Verus supports *ghost code* (e.g., functions marked with `spec` or `proof`), which is not executed at runtime but used to help reason about the program's behavior. In Vest, we leverage these ghost functions to precisely define the formal semantics and to capture the correctness and security properties of the parsers and serializers. Finally, using Verus, we develop modular and reusable specifications, proofs, and implementations within VestLib that the VestDSL compiler employs extensively as it processes user-defined formats.

Understanding the generated code. The VestDSL compiler translates each DSL construct straightforwardly to a lower-level combinator in VestLib, which serves as an intermediate representation for the underlying format. For example, under the hood, the VestDSL's structure format `a` = { x: `u8`, y: `u8` } (described in the previous section) translates to a `Pair` combinator that sequentially composes two `U8` combinators, and a `Mapped` combinator that maps the result of the `Pair` combinator to the fields x and y of the struct (and back for serialization).

For each combinator, VestLib provides a Verus specification, defining the combinator's formal semantics for both parsing and serialization, a proof of correctness and security for that spec, *and* a performant implementation that provably obeys the spec.

Crucially, VestLib's combinators have been carefully designed to have a uniform interface, which enables straightforward composition into complex formats. In other words, given a user-defined format, which is typically more complicated than a simple struct with two integer fields, the VestDSL compiler can cleanly combine VestLib combinators to build up a specification, a proof, and a corresponding implementation from the existing combinator definitions in the library. We give a multi-step example in Figure 3, which shows the output of compiling the VestDSL description of the TLS 1.3 `server_hello` format.

The `server_hello` format **(a)** specifies a sequence of fields that are either primitive formats (e.g., `u8`) or sub-formats defined elsewhere in the specification. The VestDSL compiler generates the corresponding Rust data type definition `ServerHello` **(b)**, which mirrors the structure of the format and serves as the internal structured representation of the serialized bytes. Then, the VestDSL compiler generates two functions that contain the intermediate representation of the format, using the combinators defined in VestLib. Leveraging the dual aspects of VestLib combinators—both their specifications and verified implementations—the compiler generates two closely matching versions of the intermediate representation, one specification version `spec_server_hello` **(c)** and an executable version `server_hello` **(d)**. The `spec` combinators used for composing `spec_server_hello` are written in a purely functional style for simplicity, but are too inefficient for practical implementations. Instead, the `server_hello` function consists of `exec` combinators, which are both verified and expands to performant code **(e)**, allowing for efficient, zero-copy parsing and in-place serialization.

2.3 Integration with Larger Projects

As parsers and serializers are often just one of the many components in a larger systems, Vest is designed to be easily integrated into existing Rust projects.

To avoid excessive data copying, prior verified binary parser generators required the programmer to write complex semantic actions [54], or to invoke a series of low-level validators, accessors, jumpers and readers for parsing, and low-level writers and finalizers for serialization [50]. In contrast, Vest sup-

(a)

```
server_hello = {
  legacy_session_id_echo: session_id,
  ciphersuite: cipher_suite,
  const legacy_compression: u8 = 0,
  extensions: server_extensions,
}
```

(b)

```
struct ServerHello<'a> {
  legacy_session_id_echo: SessionId<'a>,
  ciphersuite: CipherSuite,
  legacy_compression: u8,
  extensions: ServerExtensions<'a>,
}
```

(c)

```
spec fn spec_server_hello() →
SpecServerHelloCombinator {
  Mapped(Pair(spec_session_id(),
          Pair(spec_cipher_suite(),
            Pair(Refined(U8, |x: u8| x == 0),
              spec_server_extensions()))),
         ServerHelloIso)
}
```

(d)

```
exec fn server_hello() → (o: ServerHelloCombinator)
    ensures o.view() == spec_server_hello() {
  Mapped(Pair(session_id(),
          Pair(cipher_suite(),
            Pair(Refined(U8, |x: u8| x == 0),
              server_extensions()))),
         ServerHelloIso)
}
```

(e)

```
let (n, session_id) = session_id().parse(&buf)?;
let (n, cipher_suite) = cipher_suite().parse(&buf[n..])?;
let (n, legacy_compression) = U8.parse(&buf[n..]).refined(|x| x == 0)?;
let (n, server_extensions) = server_extensions().parse(&buf[n..])?;
ServerHello { legacy_session_id_echo, cipher_suite,
              legacy_compression, server_extensions }
```

Figure 3: *A VestDSL spec of the* `ServerHello` *message in TLS 1.3* ***(a)****, the corresponding data type definition in Rust* ***(b)****, the format's formal semantics* ***(c)****, the combinator-based implementation of the format* ***(d)****, and the result* ***(e)*** *of expanding the exec combinators by calling* `server_hello().parse(&buf)`. *Some details omitted for brevity.*

ports *zero-copy* parsing for byte arrays, where parsed data types (e.g., `struct ServerHello<'a>` in **(b)**) contain *references* to the input buffer, rather than copying the bytes into the data type.[2] Vest does this by leveraging Rust's *lifetimes*, which encapsulate the aliasing relationship between the parsed data type and the input buffer. Without lifetimes, we would need a complex, ad-hoc separation logic predicate for each parser to guarantee memory safety.

Ultimately, Vest provides a simple, idiomatic developer interface without compromising performance. For example, the developer can include VestLib as a dependency in their (verified or unverified) Rust project, import the generated module, and use the generated `server_hello` parser and serializer as "one-liners" in their application code:

```
let (consumed, msg1) = server_hello().parse(&ibuf)?;
// produce a new message msg2 to write back out
let len = server_hello().serialize(&mut obuf, &msg2, 0)?;
```

The `parse` function operates on byte slices (`&[u8]` in Rust); the `serialize` function writes the serialized value *in-place* to a byte buffer, given a mutable reference (`&mut Vec<u8>`) and an offset (0 in the example above), without any heap allocation. Consequently, the generated parsers and serializers are not only verified for correctness and security but are also ergonomic and efficient. With Vest, programmers can concentrate on the high-level design of their systems, while the VestDSL compiler and VestLib handle the low-level details of parsing and serialization.

[2]Small primitive types (e.g., u32, where a reference would be more memory-expensive than copying) are copied into the resulting data type.

3 Vest's Design

We present the design of Vest, a novel framework for verified binary parsing and serialization. Vest has two parts: VestDSL, a *simple* yet *expressive* front-end language for specifying binary formats (§3.1), and VestLib, our back-end library of parser/serializer combinators, which we use to give state-of-the-art *correctness*, *security*, and *performance* to VestDSL formats.

3.1 A Language for Defining Binary Formats

Previous frameworks for verified binary parsing and serialization either lack a high-level format definition language [57], provide format definitions as meta-programs embedded in proof assistants [25, 59], or lack expressivity [50]. We argue that an expressive, and user-friendly format definition language is essential for bridging the gap between informal binary format descriptions and verified parsers and serializers.

Hence we design VestDSL as a standalone language for binary formats with semantics defined via VestLib. Each VestDSL format defines a corresponding Rust data type, along with a VestLib combinator that defines how parsing and serialization should behave, along with their relevant security properties. Figure 4 shows VestDSL's main syntactic constructs, their intuitive meaning, and their semantics in VestLib.

Bytes, Integers, and Refinements. The first four rows of Figure 4 show VestDSL's first-class support for various byte arrays and integers, which are the basic building blocks of binary formats. The syntax `[u8; N]` represents a fixed-length byte array of length N. The semantics maps this construct to the `bytes::Fixed::<N>` combinator in VestLib, which handles parsing and serializing a fixed number of bytes.

	VestDSL Syntax	Description	VestLib Semantics
1	`[u8; N]`	Fixed-length byte-arrays	`bytes::Fixed::<N>`
2	`u8`, `u16`, `u24`, `u32`, `u64`	Fixed-size unsigned integers	`U8`, `U16Be`, `U16Be`, `U24Be`, `U24Le`, . . .
3	`u16 | 8..0xFFFE`	Integers with constraints	`Refined(U16Le, |x: u16| 8 ≤ x ≤ 0xFFFE)`
4	`const a: [u8; 48] = [0; 48]`	Constant byte-arrays	`Refined(bytes::Fixed::<48>, |x: &[u8]| x == &[0; 48])`
5	`b(@len: u24)= [u8; @len]`	Variable-length byte-arrays	`fn b(len: u24) → _ { bytes::Variable(len.into()) }`
6	`c = { @x: u24, y: b(@x)}`	Struct with dependent fields	`Mapped(Pair(U24Le, |x: u24| b(x)), CIso)`
7	`d = (@cnt: u64)= [a; @cnt]`	Count-bounded repetitions	`fn d(cnt: u64) → _ { RepeatN(a(), cnt.into()) }`
8	`e = { @len: u8, v: [u8; @len]` `≫= Vec<d> }`	Length-bounded repetitions	`Mapped(Pair(U8, |len|` `AndThen(bytes::Variable(len.into()), Repeat(d())), EIso))`
9	`g = Option<[u8; 32]>`	Optional values	`Opt(bytes::Fixed::<32>)`
10	`f = enum { A = 1, B = 2 }`	Finite enumerations	`Refined(U8. |x| x == 1 || x == 2 )`
11	`h(@flag: f)= choose(` `@flag){ A ⇒ c, B ⇒ e }` `i = { @tag: f, y: h(@tag)}`	Implicitly tagged and explicitly tagged unions	`fn h(flag: F) → _` `{ Mapped(Choice(Cond(flag == F::A, c()), Cond(flag == F::B, e())), HIso) }` `Mapped(Pair(f(), |tag| h(tag)), IIso)`
12	`msg = choose { A(u8 | ..0xfe),` `B(u8 | 0xff)}`	Untagged unions	`Mapped(Choice(Refined(U8, |x| 0 ≤ x ≤ 0xFE),` `Refined(U8, |x| x == 0xFF)), MsgIso)`

Figure 4: **VestDSL Syntax and Its Semantics in VestLib**. `u8`, `u16`, . . . *are primitive formats;* `a`, `b`, . . . *are user-defined format names;* `choose`, `const`, . . . *are VestDSL keywords; and* `Fixed`, `Refined`, `Mapped`, . . . *are VestLib specification combinators.* `CIso`, `EIso`, . . . *are functions that convert between anonymous, structural types and the corresponding* (iso*morphic) nominal, Rust data types.*

The `u8`, `u16`, `u32`, and `u64` types in VestDSL represent fixed-size unsigned integers, which are mapped onto their corresponding primitive types in Rust. VestDSL allows specifying the endianness of the integers (defaulting to little-endian) and automatically selects the corresponding combinator in VestLib; for example, `U16Le` in VestLib defines a parser that reads two bytes from the input buffer and interprets them as an unsigned 16-bit integer in little-endian order (and vice versa for serialization). VestDSL also supports `u24`, which is used for instance in TLS (§6.2); to support `u24` in VestLib, we use a 3-byte array internally and include *verified*, *bidirectional* conversions between this byte array and `u32`s (constrained to fall within the appropriate range).

In addition to basic integers and byte arrays, VestDSL supports defining formats with specific constraints on their values. Rows (3) and (4) in Figure 4 illustrate this with a 16-bit integer constrained to a specific range and a 48-byte array with a constant value. VestDSL maps these *refinements* of formats into the `Refined` combinator in VestLib, which checks a predicate after parsing and before serializing the value. These formats are represented by the same type as the base format, but the values must satisfy the predicate.

Parametric and Dependent Formats. VestDSL has robust support for parametric and dependent formats, which are essential for expressing binary formats whose field structures or sizes depend on other fields or external parameters. The syntax `format(@x: a)= ...@x...` defines a format with *external* dependencies, where `@x` in format `a` is passed as a parameter and used in the format definition. The semantics of parametric formats correspond to *functions*. For example, `b(@len: u24)= [u8; @len]` in row (5) defines a parametric format that contains a variable-length byte array: it semantics is a function that takes a `u24` length and returns the `bytes::Variable` combinator for parsing or serializing a dynamically known length of bytes.

Internal dependencies within formats, such as `c = { @x: u24, y: b(@x)}` in row (6), are also supported in VestDSL. In this example, the format `c` is a struct with two fields: `@x`, an unsigned 24-bit integer, and `y`, a byte array whose length depends on the value of `@x`. For all struct formats, VestDSL first maps the format to a (dependent) `Pair` of combinators in VestLib, where the `Pair` combinator parses (or serializes) the first field and then uses its value to determine the desired behavior for parsing and serializing the second field. However, as a generic combinator, `Pair` operates on anonymous tuples, so if VestDSL only used the `Pair` to encode dependent struct formats, users would have to work with the tuple type `(u24, byte_array)`, which is not very ergonomic. Instead, VestDSL also leverages the `Mapped` combinator in VestLib to convert between the anonymous tuples and the corresponding nominal struct types in Rust (e.g., a struct with fields `x` and `y` for format `c`).

Bounded Repetitions. Another common pattern in binary formats is a contiguous sequence of *homogeneous* data elements, often referred to as a *list* or *vector* format. For security and predictable parsing, these repetitive formats often need to be *bounded*, either by a preceding count field indicating the number of elements, or by a preceding length field indicating the total number of bytes occupied by the following elements. VestDSL supports both kinds of repetitions, as shown in rows ⑦ and ⑧ of Figure 4. The syntax `[a; N]` represents a fixed number of repetitions of the format, where `N` can be a constant integer or a dependent value. The semantics of this construct is defined by VestLib's **`RepeatN`** combinator, which takes a given combinator and an integer *n* and repeats the combinator *exactly n* times.

On the other hand, the syntax `[u8; @len] ≫= Vec<d>` in row ⑧ defines a length-bounded repetition. The infix operator ≫= (monadic bind) is used to *re-interpret* parsed bytes into a different format, and the **`Vec`** construct is used to define an *unbounded* repetition (via the **`Repeat`** combinator) of the format `d`. Crucially, the semantics of ≫= defined by the **`AndThen`** combinator ensures that for security, `Vec<d>` must consume exactly `@len` bytes. Both formats correspond to Rust's vector type `Vec<T>`, where `T` is the data type of the repeated format.

Variant Formats. One of the most interesting and challenging formats to support in a verified parser/serializer framework are *variant* formats. These formats often define structures that can take multiple forms. The simplest variant format is an *optional* format, as shown in row ⑨ of Figure 4. The format `g` wraps a 32-byte array with the **`Option`** construct. The semantics of this format is defined by the **`Opt`** combinator, where for parsing, it invokes the inner combinator and returns `None` if the inner combinator fails, and for serialization, it serializes the value if it exists, or does nothing if the value is `None`.

More complex variant formats include *unions*, which can be *implicitly tagged*, *explicitly tagged*, or even *untagged*. For instance, the format `h` in row ⑪ is an implicitly tagged union, where the body of the format definition uses the **`choose`** construct to dynamically select between two formats `c` and `e` based on the value of an external enumeration format `@flag`. The implicitly tagged union can be made explicit by defining a struct format that includes a `@tag` field preceding the union field, as shown in the definition of format `i`. VestDSL uses a combination of VestLib's **`Choice`**, **`Cond`**, and **`Mapped`** combinators to define the semantics of tagged unions. The (ordered) **`Choice`** combinator takes two combinators and returns the result of the first if it succeeds; otherwise, it backtracks and returns the result of the second. The **`Cond`** combinator takes a boolean expression and only invokes the given combinator if the expression evaluates to true. Lastly, the **`Mapped`** combinator applies the isomorphism between structural sum types (nested `Either`s) and Rust's nominal enum types after parsing and before serialization, exposing an ergonomic interface to users, similar to the struct format. For untagged unions like the format `msg` in row ⑫, VestDSL uses **`Choice`** and **`Mapped`** to define its semantics in a similar manner.

It is important to note that variant formats are only *conditionally* secure. For example, if we modify row ⑨ to **`Option`** `<[u8; 0]>` or row ⑫ to **`choose`** `{ A(u8 | 0..7), B(u8 | 5..10)}`, neither format remains secure. We discuss the formal security of such formats in §4, and provide additional examples in §6.

3.2 A Compositional Combinator Library

We now turn to VestLib, a library of verified parser/serializer combinators in Verus. Aside from providing the semantic underpinning of VestDSL, expert developers may also use VestLib directly to implement new combinators. All VestLib combinators are designed to be *high performance*: both in generated concrete implementations and verification time.

We implement our combinator library using Rust/Verus's excellent support for *traits*. In Rust, traits define a set of methods and associated types that types must implement, similar to interfaces or type classes in other languages. For example, one can define a trait `Parser` with an associated type `Output` and a method `parse` that takes a byte slice and returns a value of type `Self::Output`.

```
trait Parser {
  type Output;
  fn parse(&self, input: &[u8]) → Self::Output;
}
```

By defining combinators as first-order, standard types (Rust's **`struct`** type) that implement certain traits, VestLib avoids the significant complexity attached to formally verifying data structures containing arbitrary higher-order functions. Instead, theorems proven for one of Vest's combinators can be *automatically* inherited by any combinator that composes with it, significantly reducing the proof burden and verification time.

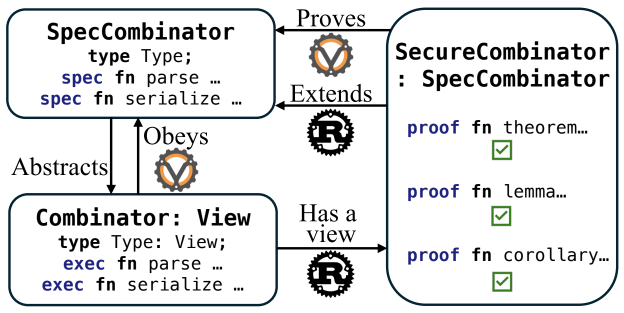


Figure 5: **VestLib's Trait-based Architecture**.

As described in §2, VestLib's design seeks to separate a combinator's specification, implementation, and correctness proofs. We achieve this through a trait-based interface with three main traits: `Combinator`, `SpecCombinator`, and `SecureCombinator`. Figure 5 illustrates their interactions. A `SpecCombinator` is any type that implements the `SpecCombinator` trait, which defines the *specification* (a mathematical view) of a combinator. It includes Verus's **`spec`**-mode *ghost* functions `parse` and `serialize`, as well as an associated type `Type` that represents the abstract data type of the format. In contrast, a `Combinator` defines the concrete data type of the format, as well as the actual, efficient implementations of `exec`-mode functions for parsing and serializing, which are proved to match the behavior of the corresponding `SpecCombinator`. A `SecureCombinator` is a sub-trait of `SpecCombinator` and defines a set of correctness and security properties that must be proven for the `SpecCombinator` to be considered secure.

Notably, each `Combinator` must implement the `View` trait, and through trait bounds at the Rust level, we force this view to be a `SecureCombinator`. This design has two key benefits. First, it makes VestLib developers *specify* the behavior of an executable combinator, as failing to do so will result in a Rust compiler error indicating that the `View` trait is not implemented. Second, it ensures that the specifications provided by VestLib developers are correct and secure by requiring machine-checked proofs of the properties defined by the `SecureCombinator` trait. By eschewing higher-order functions, our trait-based design provides fast verification times, allowing developers to quickly iterate through a "specify-implement-verify" workflow. More detail about our combinator design is given in Appendix A.

4 Vest's Correctness & Security

Central to Vest is its strong correctness and security guarantees, which lie along three axes:

- **Functional Correctness and Memory Safety**. [3] All Vest executable parsers/serializers are proven equivalent to their specs (§3.2), written in simple, pure functions over mathematical sequences of bytes. Functional equivalence precludes crashes, non-termination, memory unsafety, and integer overflows.
- **Parser & Format Security**. Our `proof fn`s in §3.2 prove strong *security lemmas* for each combinator (§4.1). Essentially, each parser and serializer must be *mutually inverse*, which eliminates large classes of vulnerabilities, e.g., format-confusion [59] and malleability [50] attacks.
- **Side-Channel Resistance**. A key novelty of Vest is its support for *side-channel resistance*, which prevents parsers that operate over secret data from leaking it through timing channels or memory access patterns. We employ *type abstraction* to generically protect parser combinators against side-channel attacks (§4.2).

4.1 Formal Correctness of Binary Formats

We outline the main correctness and security guarantees of all VestLib combinators. As identified in prior work (e.g., Comparse [59]), we need parsers and serializers to be *mutually inverse*. Requiring parsing and serialization to be inverse rules out large classes of vulnerabilities, as we discuss below.

[3] Vest does not use **`unsafe`** Rust.

To formalize our mutual inverse properties, we first formally define a binary format and its constituent routines. Let $\mathbb{B} = \{0, \ldots, 255\}^*$ denote the set of byte sequences. Then, a binary format $\mathbf{F}$ is a tuple $(\mathbb{T}, \mathbf{P}, \mathbf{S})$, where: $\mathbb{T}$ is an abstract data type; $\mathbf{P} : \mathbb{B} \to (\mathbb{T} \times \mathbb{N}) + \perp$ is a partial parsing function mapping bytes to a representation in $\mathbb{T}$, along with the number of bytes parsed; and $\mathbf{S} : \mathbb{T} \to \mathbb{B}$ is a total serializing function that returns a byte sequence. Parsers may fail and return $\perp$ due to unexpected or malformed inputs (e.g., not enough data available for parsing).

We formally state our mutual inverse properties via two Round Trip theorems. We prove these Round Trip theorems for all Vest formats, including our case studies (§6).

Theorem 1 (1st Round Trip Theorem)**.** *For every binary format $\mathbf{F} = (\mathbb{T}, \mathbf{P}, \mathbf{S})$, if $\mathbf{S}(v)$ returns $b \in \mathbb{B}$, then $\mathbf{P}(b) \neq \perp$ and returns $(v, |b|)$, where $|b|$ is the length of b.*

Intuitively, Theorem 1 says that if serializing a structured value gives a byte sequence, then parsing that byte sequence must succeed and return the original value. This defines correctness of the parser with respect to the serializer.

Theorem 2 (2nd Round Trip Theorem)**.** *For every binary format $\mathbf{F} = (\mathbb{T}, \mathbf{P}, \mathbf{S})$, if $\mathbf{P}(b) \neq \perp$ and returns (v, n), then $\mathbf{S}(v) \neq \perp$ and returns $b[0, \ldots, n]$.*

Similarly, Theorem 2 states that if parsing succeeds, then serializing the result must give back the same byte buffer (up to leftover bytes from parsing). This defines correctness of the serializer with respect to the parser.

Crucially, both theorems are strong enough to derive several important *security* properties for binary parsers and serializers. For instance, from the 1st Round Trip Theorem, we prove that a serializer is *injective*, meaning that distinct structured values are always serialized into distinct byte sequences. This property ensures that each byte sequence uniquely represents a structured value, preventing format ambiguities that could lead to format confusion attacks [59]. Similarly, from the 2nd Round Trip Theorem, we prove that a parser is *non-malleable*, meaning that a byte sequence cannot be altered in a way that misleads the parser. This ensures that every well-formed structured value has a unique byte sequence representation, preventing a range of parser malleability attacks [50, 54].

Auxiliary Lemmas. To prove our round trip properties, we rely on several auxiliary lemmas for each combinator. First, following prior work [50, 59], we prove that certain parsers satisfy a *prefix lemma*. This lemma asserts that if a byte sequence b parses to a value v, then any extended sequence $b + b'$ must also parse to v with the same number of bytes parsed. Prefix lemmas are essential for parsing sequentially composed formats (e.g., structs and repetitions), as the parser must recognize when one field ends and the next begins.

Next, we prove that certain parsers are *disjoint* from others, ensuring non-ambiguity in untagged unions (§3.1, §6.3). We again leverage Rust's trait system for *compositional verification*; by generically implementing disjointness lemmas for higher-order combinators (e.g., `Pair`, `Choice`), we can automatically derive disjointness for user-defined combinators.

Finally, we prove each combinator's *productivity*, ensuring that successful parsing or serialization consumes or produces at least one byte. Similar to the theory of derivatives for regular expressions [48], we require a combinator to be productive whenever we apply it *zero-or-more* times (e.g., `Opt`, `Repeat`).

4.2 Side-channel Security

Motivation. Defending against side-channel attacks is particularly important for parsers and serializers that handle secure data. For example, consider a virtual private network (VPN) protocol in which two peers exchange encrypted IP packets over a public network. When an encrypted packet arrives, the recipient must parse the decrypted data according to the IP packet format; if the parser's execution time depends on parsed values, then it could disclose information about the secret IP traffic. Attacks of this flavor have been demonstrated against the Zcash blockchain [56] and the mbedTLS library [6], underscoring the importance of side-channel-secure parsing and serialization.

Our Approach. Vest protects against classic digital side-channel attacks by supporting a form of *type abstraction*, similar to prior work [62]. This technique encapsulates secret data inside of an opaque wrapper type, where application code cannot use values of that type for control flow or memory accesses. Our wrapper type prevents surrounding code from accessing the values of individual bytes, but does *not* prevent code from reading the size of protected bytestrings or from slicing those bytestrings into multiple parts. In this way, our side-channel resistance properties guarantee privacy for the *values* of data but not their lengths.

To construct this opaque wrapper type, one could naively hardcode a particular wrapper inside of Vest, e.g., one named `SecretBytes`. While this would suffice for simple use cases, advanced applications may need custom security policies (e.g., *declassification* [24] policies); these would necessitate custom opaque wrapper types that implement the desired policy. Hardcoding `SecretBytes` within Vest would preclude applications from defining their own wrapper type (at least, without forking Vest), effectively forcing all users of Vest to adhere to whatever security policy we define.

In contrast, our approach to side-channel security does not fix a particular "secret" type, but instead exposes an *interface* on buffers that specifies which operations are needed to parse and serialize. Thus, Vest can provide constant-time parsing for exactly the formats that support it, i.e., those that do not rely on reading specific concrete values during parsing. Users can define their own opaque types according to their desired security policy—the types need only implement our interface to work with Vest.

We define our buffer interface via Rust traits. The strictest interfaces are given by the traits `VestInput` and `VestOutput`, which only allow one to retrieve the buffer's length and set/retrieve subranges of buffers. For parsers and serializers that *require* viewing the data (e.g., converting between two bytes and a `u16`), we have the interfaces `VestPublicInput` and `VestPublicOutput`, which *extend* their corresponding strict interfaces. These interfaces are for fully public bytes, since they allow the code freely convert between the underlying buffer and a Rust slice of `u8`s. The full trait definitions for our buffer interface are provided in Appendix A.

Use Cases. VestDSL defaults to generating parsers and serializers that operate over `VestPublicInput` and `VestPublicOutput` types. This allows VestDSL to support rich binary formats that contain primitive integer types and arbitrary dependencies (which may necessitate inspecting the input data when parsing). For side-channel resistant formats, developers can directly use VestLib and define the custom `SecretBytes` type discussed above. The `SecretBytes` type may implement only the `VestInput` and `VestOutput` traits, hiding the underlying buffer permanently, or it may also allow the user to retrieve the value of the buffer at a particular index, subject to user-defined declassification constraints.

Caveats. Although attempts to introduce side-channel vulnerabilities in our implementation, e.g., by branching on or accessing memory using a secret value while parsing, are immediately rejected by `rustc`, Vest does not provide a formal statement of side-channel resistance, as Verus cannot directly express properties about execution time. Additionally, as discussed in §1, Vest provides side-channel resistance only at the Rust source level; we trust `rustc` not to introduce control-flow or memory-access dependencies that violate our type abstraction. Formalizing a cost semantics [19] and writing a constant-time preserving compiler [15, 23] for Rust remains an open problem.

5 Vest's Implementation

VestLib includes ten primitive combinators, nine higher-order combinators, and five derived combinators. Each combinator comes with a specification, implementation, and formally verified correctness and security properties. In total, this amounts to ∼6K lines of Verus code (with ∼1:1:1 ratio of `spec`, `proof`, and `exec` code). Notably, VestLib does not rely on any external Rust libraries (other than Verus), and every component is formally verified, ensuring there are no unverified, trusted components. With our carefully designed, trait-based interface for combinators and Verus's excellent verification performance, the entire library verifies in less than 10 seconds.

We develop the VestDSL compiler in 5,807 lines of Rust code (including 1,240 for parsing, 1,442 for elaboration and type checking, and 3,125 for code generation). As VestDSL is designed as a standalone language, we include features to allow users to define formats in a succinct and precise manner. These features include a basic macro system, exhaustiveness checks for the `choose` construct, cyclic format checks, type inference for enumerations, and type checking for dependent and parametric format definitions and invocations. To provide users with an ergonomic interface for the generated data types, we adhere to user-provided format names, field names, and variant names, generating idiomatic Rust type definitions.

6 Case Studies

We present three case studies to demonstrate Vest's capabilities: the Bitcoin block and transaction formats, TLS 1.3 handshake message formats, and the WebAssembly binary format. We choose these binary formats because they are widely used in practice, have complex structures, and are designed for high-performance and security-critical applications. Although there are existing verified [50] or *partially* verified [21, 50] parsers and serializers for these formats, we demonstrate that Vest offers greater expressivity and orders of magnitude faster verification.

Producing the necessary format definitions in VestDSL required careful work reading through existing documentation and occasional corrections to achieve compatibility. Nonetheless, using lines of code as a crude metric, we can compare the size of the baseline, manually implemented parsers and serializers to the size of the VestDSL format descriptions and find that VestDSL provides substantial productivity gains. Specifically, the baseline Bitcoin library is ∼2,000 LoC (vs. ∼67 LoC in VestDSL), TLS is ∼7,000 LoC (vs. 500), and WebAssembly is ∼30,000 LoC (vs. 600 LoC).

6.1 Bitcoin Block and Transaction Formats

A Bitcoin block consists of a block header and a variable number of transactions. Parsing and serializing the block header is fairly simple, as it is a fixed-size sequence of 80 bytes containing fields that are all built-in formats in Vest: the version (`u32`), previous block hash (`[u8; 32]`), Merkle root (`[u8; 32]`), timestamp (`u32`), target (`u32`), and nonce (`u32`). In contrast, the Bitcoin transaction format is more complex, particularly with the segregated witness (segwit) extension [42], which prior work like EverParse [50] cannot handle out-of-the-box.

The left of Figure 6 shows the binary structure of a Bitcoin transaction with the segwit extension, with a description of each field. Crucially, the format contains an *optional* witness flag after the version field and a list of witnesses whose value is dependent on the witness flag and the input transactions. This mix of interleaved dependencies and variant formats makes the transaction format challenging to define.

Bitcoin Variable-length Integer. To specify the Bitcoin transaction format, we first add a primitive format `btc_varint` in VestDSL that represents the Bitcoin variable-length integer [17], which uses a space-efficient encoding illustrated in Figure 7. These integers are used to represent the number of transaction inputs, outputs, and witnesses. We implement `btc_varint` using VestDSL, as shown in the `btc_varint_helper` definition below, except for a custom conversion function that maps between the internal data type of `btc_varint_helper` and the appropriate Rust integer type. Importantly, we implement the custom conversion with VestLib combinators, ensuring that `btc_varint` enjoys the same correctness and security guarantees as other Vest formats.

```
btc_varint_helper = {
  @tag: u8,
  rest: choose(@tag) {
    0..0xFC ⇒ empty,
    0xFD ⇒ u16 | 0xFD..0xFFFF,
    0xFE ⇒ u32 | 0x10000..0xFFFFFFFF,
    0xFF ⇒ u64 | 0x100000000..,
  },
}
```

Bitcoin Transaction Format in VestDSL. Using `btc_varint`, we can now specify the Bitcoin transaction format in VestDSL, shown in Figure 6. This format is tricky to specify because of the optional witness field and the associated witness data. We identify three subtle points from the informal description of the Bitcoin transaction format from the Bitcoin wiki. First, the witness `flag`, if present, is always two consecutive bytes [0x00, 0x01]. Second, the number of transaction inputs, encoded as a `btc_varint`, is never zero. Third, the witness count is equal to the transaction input count, and the witness data is omitted if the witness `flag` is absent.

Given these points, we specify the Bitcoin transaction format as a choice of two sub-formats: one for the newer, segwit-enabled case and one for the legacy, non-segwit case. The parser of the `tx` format first parses version as a `u32`, then parses @txin_count as a `btc_varint`, and finally `choose`s between the segwit and non-segwit sub-formats based on the value of @txin_count. If @txin_count is 0x00, the `tx_segwit` format is chosen, where 0x00 is interpreted as the first byte of the witness flag; otherwise, the `tx_nonsegwit` format is chosen. `tx_segwit` contains the remaining byte of the witness flag, the transaction inputs, the transaction outputs, the witnesses, and the lock_time, while `tx_nonsegwit` simply takes the number of transaction inputs as a parameter and omits the witnesses. Importantly, txins and witnesses in `tx_segwit` are both bounded by the number of transaction inputs @txin_count, satisfying the third point above.

6.2 TLS 1.3 Handshake Messages

TLS 1.3 is a widely adopted cryptographic protocol that ensures secure communication over the internet by providing confidentiality, integrity, and authenticity of data exchanged between clients and servers. The security of TLS 1.3 relies on the correct implementation of its underlying binary formats, which are detailed across numerous RFCs [14, 16, 29, 31, 39, 45, 47, 51, 60]. Crucially, the high-level message serialized by the sender must match the value parsed by the recipient, which is satisfied by the mutual inversion property guaranteed for all parsers and serializers gener-

Size	Data Type	Descriptions and comments
4	`uint32_t`	Transaction format `version` number
0/2	`optional uint8_t[2]`	Witness `flag`. If present, always 0001, indicates a segwit transaction
1+	`var_int`	`txin_count`, # of trans. inputs (> 0)
41+	`tx_in[]`	List of 1 or more trans. inputs
1+	`var_int`	`txout_count`, # of trans. outputs
9+	`tx_out[]`	List of 1 or more trans. outputs
0+	`tx_witness[]`	List of witnesses, 1 for each `tx_in`; omitted if the witness `flag` is absent
4	`uint32_t`	Block number or timestamp at which this transaction is unlocked

```
tx = {
  version: u32,
  @txin_count: btc_varint,
  rest: choose(@txin_count) {
    0x00 ⇒ tx_segwit,
    _ ⇒ tx_nonsegwit(@txin_count),
  },
}
```

```
tx_segwit = {
  const flag: u8 = 1,
  @txin_count: btc_varint,
  txins: [txin; @txin_count],
  @txout_count: btc_varint,
  txouts: [txout; @txout_count],
  witnesses: [witness; @txin_count],
  locktime: lock_time,
}
```

```
tx_nonsegwit(@txin_count: btc_varint) = {
  txins: [txin; @txin_count],
  @txout_count: btc_varint,
  txouts: [txout; @txout_count],
  locktime: lock_time,
}
```

Figure 6: **Bitcoin Transaction Format**. *As described in the Bitcoin wiki [18] (left) and the VestDSL specification (right).*

Size	Format
1	A `uint8_t` less than 0xFD
3	0xFD followed by a `uint16_t` in [0xFD, 0xFFFF]
5	0xFE followed by a `uint32_t` in [0x10000, 0xFFFFFFFF]
9	0xFF followed by a `uint64_t` greater than 0xFFFFFFFF

Figure 7: **Bitcoin Variable-length Integer Format [17]**. *An integer less than 0xFD is encoded as an 8-bit integer. Larger integers are encoded as a 1-byte prefix indicating the integer length followed by the (little endian) integer.*

ated from VestDSL. In this section, we highlight key aspects of the TLS 1.3 handshake message formats that are prone to implementation errors and demonstrate how VestDSL can avoid such pitfalls.

Handling Tag-Length-Value (TLV) Formats. TLV formats are commonly used in TLS 1.3 handshake messages to encode data types. Each TLV element consists of a type (tag) identifier, the length of the data, and the value itself. This structure allows for flexible and extensible data encoding but also introduces potential pitfalls, such as incorrect length calculations and improper handling of nested TLV elements. Figure 8 illustrates the VestDSL definition of the TLS 1.3 handshake message format. The `handshake` parser first parses two dependent fields sequentially: the message type @msg_type (as a `handshake_type`, an enumeration format) and the @length (as a `u24`, a 3-byte unsigned integer). It then extracts @length bytes from the input buffer as the payload and parses this payload (instead of the original input buffer) using a `choose` construct based on @msg_type. Crucially, the $\gg=$ operator in VestDSL ensures that the sub-format *fully consumes* the `payload` field, adhering to the requirements specified in the RFC. Additionally, since the $\gg=$ operator can be chained within the definition of sub-formats, it ensures this length restriction *transitively* for any nested TLV-encoded messages.

```
handshake = {
  @msg_type: handshake_type,
  @length: u24,
  payload: [u8; @length] ≫= choose(@msg_type) {
    ClientHello ⇒ client_hello,
    ServerHello ⇒ sh_or_hrr,
    NewSessionTicket ⇒ new_session_ticket,
    EndOfEarlyData ⇒ empty,
    EncryptedExtensions ⇒ encrypted_extensions,
    Certificate ⇒ certificate,
    CertificateRequest ⇒ certificate_request,
    CertificateVerify ⇒ certificate_verify,
    Finished ⇒ finished(@length),
    KeyUpdate ⇒ key_update,
  },
}
```

Figure 8: **TLS 1.3 Handshake Message in VestDSL**.

Handling Implicitly Tagged Union Formats. Implicitly tagged unions are formats whose structure is determined by external (or contextual) information, rather than an explicit tag, easily leading to ambiguities and implementation errors. For example, the `finished` message shown in Figure 8 is an implicitly tagged union where the message type is determined by the digest size of the hash algorithm negotiated during the handshake. We can easily express this using VestDSL's parametric formats:

```
finished(@size: digest_size) = choose(@size) {
  HashLegacy ⇒ [u8; 12],
  Sha256 ⇒ [u8; 32],
  Sha384 ⇒ [u8; 48],
  Sha512 ⇒ [u8; 64],
  _ ⇒ [u8; @size],
}
```

In contrast, tools like EverParse [50], which do not support parametric formats, must either express implicitly tagged unions as uninterpreted bytes (essentially not parsing the format) or parse them in a *staged* fashion, which is not compositional and requires the user to manually invoke the correct parser/serializer based on the external information.

6.3 WebAssembly Binary Format

WebAssembly (Wasm) [32] is a binary instruction format that serves as a portable compilation target for languages like C, C++, and Rust. Wasm is designed both for fast decoding and execution and for safety and security, making it suitable for high-performance applications that execute in untrusted environments. Naturally, these benefits depend on the format decoder and encoder implementations being secure, correct, and efficient, which is precisely what Vest offers.

Capturing Common Patterns with Macros. The official Wasm binary format specification [13] describes some of its formats using grammar rules that involve "meta-variables". For example, the `section` grammar rule is defined as a constant byte (section ID) followed by an unsigned integer and some arbitrary grammar rule.

$$\text{section}_N(B) ::= \texttt{N:byte}\ \mathit{size}{:}u32\ \mathit{cont}{:}B\ (\|B\| = \mathit{size}) \mid \varepsilon$$

Here `N` can be instantiated with concrete byte values and `B` can be instantiated with any grammar rule that would consume *size* number of bytes. Since there are more than 100 formats in the Wasm spec, manually expanding such meta-variables into concrete formats in VestDSL would be tedious and error-prone. For such cases, VestDSL supports user-defined macros, allowing us to mirror the Wasm spec. For example, the `section` rule above can be simply expressed and reused in VestDSL as follows:

```
macro section!(n, t) = {               memsec = section!(5, mem)
  @size: wrap(u8 = n, u32),            globsec = section!(6, global)
  cont: [u8; @size] ≫= t,              expsec = section!(7, export)
}                                      ...
```

Securely Defining Variant Formats. Wasm includes numerous *variant* formats that represent choices between different language constructs. As discussed in §3.1 and §4.1, variant formats like `Choice` or `Opt` are only *conditionally* secure. We define these variant formats precisely as specified in the Wasm documentation and rely on Vest to verify the resulting parser and serializer and confirm that the Wasm spec is secure.

For instance, Wasm defines the `valtype` rule as a choice between `numtype`, `reftype`, and `funcref`, expressed in VestDSL using the choose construct. The generated parser and serializer can only be secure if the `numtype`, `reftype`, and `vectype` formats are *mutually disjoint*. As these three formats are indeed defined as distinct enumerations in the Wasm spec, Verus verifies the disjointness conditions Vest generates.

As another example, all instantiated sections discussed above are defined as *optional* formats in the Wasm specification, which means that a module with zero sections present and a module with all sections present are both valid. This kind of format is non-ambiguous only if each of the sections is *productive*—i.e., non-empty. Again, the Wasm spec requires that all sections, if present, must be non-empty (with an explicit section ID), and Vest formally verifies this property.

Securely Omitting Parsed Data. Decoding is just the first step in executing a Wasm module, so it is important to provide a clean interface for the resulting data types to make them easy to use in the rest of the Wasm interpreter or compiler. For instance, the import description format `importdesc` is defined as a tagged union of four different types, which should ideally be parsed into a high-level sum type that abstracts away tag details and allows for pattern matching, as shown below:

$$\begin{array}{llcll} \text{importdesc} ::= & 0x00\ \text{typeidx} & \Rightarrow & \text{importdesc} ::= & \text{fun typeidx} \\ & \mid\ 0x01\ \text{tabletype} & & & \mid\ \text{tab tabletype} \\ & \mid\ 0x02\ \text{memtype} & & & \mid\ \text{mem memtype} \\ & \mid\ 0x03\ \text{globaltype} & & & \mid\ \text{glb globaltype} \end{array}$$

Unfortunately, existing verified binary parsing and serialization frameworks do not support *omitting* parsed data and must expose low-level binary format details to the rest of the system. With Vest, however, we can define the `importdesc` format as a choice between four wrapped subformats, generating a high-level, easy-to-use `enum` type in Rust:

```
importdesc = choose {                       1 enum Importdesc {
  Fun(wrap(u8 = 0, typeidx)),               2   Fun(Typeidx),
  Tab(wrap(u8 = 1, tabletype)),             3   Tab(Tabletype),
  Mem(wrap(u8 = 2, memtype)),               4   Mem(Memtype),
  Glb(wrap(u8 = 3, globaltype)),            5   Glb(Globaltype),
}                                           6 }
```

Importantly, allowing users to omit fields *arbitrarily* can lead to ambiguities, so VestDSL restricts data omission to the `wrap` construct, ensuring that omitted fields are constant, statically known values. In this way, Vest remembers the omitted fields and correctly encodes each variant during serialization.

Limitations. Wasm represents structured control flow instructions (`block`, `loop`, and `if`) as nested, recursive formats. However, decoding such recursive structures can be resource-intensive and may cause stack overflow issues even for moderately-sized modules [4]. To address this, we follow the approach of existing Wasm binary parsers [4, 12] and parse the instructions as a flat vector. One caveat is that the opcode `0x0B` is used both for control flow terminators and to indicate the end of the instruction stream. Since Vest does not have a concept of a "stack counter" we cannot validate the well-bracketedness of nested instructions directly. Instead, we use a preprocessor to encode the size before each instruction stream. This limitation is also present in prior verified WebAssembly parsers, such as vWasm [21], which uses EverParse [50] and employs a more complex preprocessor to handle instructions.

Bitcoin	**Vest**	RustBitcoin
Parsing (ms)	253.6	593.9
Serializing (ms)	252.1	93.7
TLS	**Vest**	Rustls
Parsing (μs)	76.6	95.7
Serializing (μs)	18.4	41.9
WebAssembly	**Vest**	Cranelift
Parsing (ms)	8.8	8.2

Figure 9: **Runtime Performance Comparison Between Vest and Unverified Baselines**. *Smaller is better.*

7 Evaluation

We aim to evaluate Vest against existing parsers and serializers to answer the following questions:

Q1 Does Vest generate parsers and serializers with competitive performance compared to state-of-the-art verified/unverified implementations (§7.1)?

Q2 How does the verification performance of Vest compare to prior verified parsers and serializers (§7.2)?

In answering both questions, we use a machine with an Intel Core i9-13950HX CPU and 32 GiB of RAM. To reduce noise, we disable hyper-threading and isolate a performance core pinned to the highest frequency.

7.1 Runtime Performance

We evaluate the performance of Vest-generated parsers and serializers with a real-world benchmark for each case study.

- Bitcoin 1K: 1,000 uniformly sampled blocks (out of ~870,000 blocks at the time of writing) from the Bitcoin main chain, with about 670 MB of data.
- TLS/Tranco: Handshake traces of TLS connections from making HTTPS requests the top 100 most visited domains according to the Tranco list [40].
- PolyBenchC [9]: A canonical Wasm benchmark [32] consisting of 30 C programs compiled to Wasm.

On these benchmarks, we evaluate Vest as well as parsers and serializers from three other state-of-the-art, hand-optimized libraries: Rust Bitcoin [10], Rustls [11], and `wasmparser` [12] (used in Cranelift [7]). All benchmarks are performed via the Rust Criterion benchmarking library [8], with a sample size of 100 and a measurement time of 5 seconds. We did not test serialization performance for WebAssembly, as `wasmparser` does not have a serializer. Note that `wasmparser` is a lazy parser that generates events for each AST node without returning an entire parsed AST. While laziness can benefit performance and be useful in certain cases, popular Wasm toolchains typically need the full AST for further compilation/analysis. Hence, we construct a full AST from the events generated by `wasmparser` to ensure a fair comparison with Vest's eager parser.

	LoC (w/o comments)				**Verif. Time**	
	VestDSL	**QD**	**Rust**	**C**	**Vest**	**EverParse**
Bitcoin	67	27	2,185	1,344	2.7 s	121 s
TLS	502	919	17,425	192,229	43.9 s	4 h
Wasm	569	501	16,836	45,402	52.7 s	29 m

Figure 10: **Vest Case Studies**. *Left to right, we show: the number of lines of VestDSL code; the number of lines of QuackyDucky (QD) code (the frontend of EverParse [50]); the total number of lines of generated Rust code by the VestDSL compiler; the total number of lines of generated C code by QuackyDucky; and the wall-clock time for Verus to verify Vest generated code vs F* to verify EverParse generated code.*

Figure 9 summarizes our results. Parsers generated by Vest perform on par or better than the unverified baselines. Vest's Bitcoin parser outperforms Rust Bitcoin by over 2.3×, Vest's TLS parser outperforms Rustls by 1.25×, and Vest's WebAssembly parser is comparable in performance to Cranelift's parser. Vest's serializers are also competitive, beating Rustls by 2.3×, although lagging behind Rust Bitcoin by 2.7×.

Vest's parsers are faster than Rustls and Rust Bitcoin because they perform fewer copies and runtime checks (e.g., arithmetic overflow checks, termination checks, and most bounds checks are provably eliminated). Vest's TLS serializer is faster than Rustls because it writes to a pre-allocated buffer in place; in contrast, since it lacks the ability to (provably) predict the size of the serialized output, Rustls must allocate a new buffer for each serialization and extend it as needed. On the other hand, Rust Bitcoin's serializer is faster than Vest's because it hand-optimizes the structure of nested vectors used for the SegWit extension.

7.2 Verification Performance

Besides runtime performance, another important metric for the usability of Vest is its verification performance, i.e., the time it takes to verify the generated parser and serializer implementations. This directly correlates to the productivity of using Vest, as a faster verification time allows the developer to discover issues early and iterate more on the design or proofs.

We compare the verification time of Vest (Figure 10) against three other formally verified implementations all written in the EverParse framework [50]: Bitcoin and TLS formats from the original EverParse work, and the WebAssembly format from vWasm [21]. EverParse Bitcoin does not support the SegWit extension (while Vest does), and EverParse verifies both TLS 1.2 and 1.3 (while Vest only verifies TLS 1.3).

In a single thread, Vest's combined verification time for both parsers and serializers beats EverParse's Bitcoin parser by 44×, EverParse's TLS parser by 328× , and vWasm's WebAssembly parser by 33×. Thus, Vest improves on the verification time of prior work by orders of magnitude.

We attribute Vest's fast verification performance to several factors. First, Vest is entirely implemented in Verus, which is designed with verification performance in mind. For instance, by utilizing Rust's ownership type system, Verus eliminates the need for the extensive heap reasoning and complex memory safety proofs found in other verification tools such as Low* (used by EverParse for zero-copy parsers).

Second, Vest's trait-based combinators (§3.2) significantly reduce the need for complex quantifier reasoning over higher-order functions. Combined with Verus's deliberately conservative use of quantifiers, this approach enhances both the performance and stability of SMT queries.

Finally, the design of the VestDSL compiler plays a crucial role in verification performance. In an earlier version of Vest, we noticed that the performance of type checking and verification in Verus degraded drastically as the complexity of the combinators grew (e.g., in deeply nested choice combinators), which was due to the interaction between complex trait bounds in Vest combinators. To mitigate this, we introduced an optimization in the VestDSL compiler to *recursively* wrap sub-combinators in a fresh combinator *without* trait bounds, which essentially caches Rust's trait resolution of sub-combinators and improves the performance of both the Rust type checker and Verus on the generated code. This optimization improved the performance of verification by over 60×.

Maintainability/Extensibility. Originally, Vest's parsers and serializers had much lower runtime performance compared to the unverified industrial baselines. This was mainly because our initial focus on verification led us to use *owned* types for both parsing and serialization, which simplified reasoning but resulted in significant memory overhead. Subsequently, we implemented and verified several optimizations on both interfaces that significantly enhanced runtime performance, without sacrificing correctness or security. Thanks to our unified combinator interface, these improvements typically required only one or two person-weeks of work (including both implementation and verification).

While a full evaluation of extensibility would require a complex user study, anecdotally, adding a new combinator to prior verification frameworks [50, 54] required significant time from deep experts in those frameworks, whereas a graduate student with prior Verus experience, but no experience with Vest, used VestLib's combinator interface to add 57 new project-specific combinators in less than 3 weeks.

8 Related Work

While there are numerous industry tools for generating binary-format parsers and/or serializers [1–3, 5, 30], here we focus on verified frameworks and how their features compare with Vest's. As context, EverParse [50] and EverParse3D [54] use the Low* [49] subset of F* [53] and then extract C code. Narcissus [25] is verified in Coq [55] and Comparse [59] is verified in F*, but both extract OCaml code. van Geest and Swierstra (vGS) [57] use Agda [22] for verification and extract Haskell code. In contrast, Vest directly produces Rust code with Verus [38] annotations. All frameworks produce parsers; all but EverParse3D produce serializers.

	Vest—This Paper	EverParse [50]	3D [54]	Comparse [59]	Narcissus [25]	vGS [57]
Parse+Serialize	●	●	◑	●	●	●
DSL	●	●	●	◑	◑	○
Dependencies	●	◑	●	●	●	●
Parameters	●	○	◑	●	○	●
Repetition	●	●	◑	◑	◑	◑
Variants	●	◑	◑	◑	◑	◑
Non-malleable	●	●	●	●	○	○
Non-ambiguity	●	●	○	●	●	●
Fast Execution	●	●	●	○	○	○
Fast Verif.	●	○	○	●	●	●
No Dbl Fetch	○	○	●	○	○	○
Side-channels	●	○	○	○	○	○

● – Provides ○ – Does not provide ◑ –Partially provides

Figure 11: Comparison of Vest and other verified frameworks.

Below, we compare these frameworks based on the features they and we consider important for expressive and secure parsers and serializers. Figure 11 summarizes the comparison.

DSL. Instead of a dedicated DSL, Narcissus and Comparse embed their DSLs in a proof assistant, which requires users to learn a fair bit of syntax specific to that proof assistant. These DSLs also tend to be less succinct than VestDSL; e.g., for TLS 1.3 Vest uses ∼500 LoC vs. ∼1,000 for Comparse.

Dependent Formats. The format definition DSL in EverParse has limited support for data-dependency; e.g., a union must have its tag immediately preceding it, precluding interleaved data dependencies. However, the underlying LowParse library does support arbitrarily complex data dependencies (with a similar dependent pair combinator), which is leveraged by EverParse3D.

Parametric Formats. EverParse and Narcissus lack any support for parametric formats, while EverParse3D only supports *value*-parametric formats (e.g., "casetypes" for implicitly tagged unions). Specifically, EverParse3D only generates *validators* for each format, requiring the user to supply additional imperative *actions* for parsing the data. In contrast, Vest directly produces complete parsers. The trade-off is that EverParse3D offers more fine-grained control over the parsing process (such as avoiding unnecessary allocations), while Vest offers more robust support for parametric formats (allowing inter-dependencies for all format definitions), albeit with slightly less flexibility.

Repetitive Formats. Only Vest and EverParse support both length-bounded and count-bounded repetitions; the others only support length-bounded repetitions.

Variant Formats. Only Vest supports `Choice` and `Opt` formats with full generality; i.e., their conditional security is manifested *formally* with our disjointness & productivity lemmas §4.1. Comparse briefly mentions disjointness and non-emptiness (which relates to productivity) in the context of mathematical relations on binary formats, but it does not formally connect them to concrete parsers and serializers. The underlying LowParse library for EverParse and EverParse3D has the notion of a "non-zero-byte" parser kind, which is similar to our productivity condition, but it does not relate it to the serializers.

Other frameworks support limited forms of variant formats. For example, EverParse and Comparse both rely on F⋆'s built-in dependent types and the `match` statement to encode tagged unions. The result is that **(1)** they cannot express (the secure subset of) untagged unions, and **(2)** they lack *compositional verification* for variant formats. Vest's `Choice` combinator builds up the proofs modularly (pairwise for the variants), whereas EverParse and Comparse need to generate ad-hoc `match` statements for each different variant format (which also tends to reduce verification performance).

Non-malleability & Non-ambiguity. Vest, Comparse, and EverParse all prove the bi-directionality of their parsers and serializers. EverParse3D only proves non-malleability (one direction of the roundtrip theorems), since it only supports parsers, while Narcissus and vGS only prove non-ambiguity. To be fair, non-malleability is not a goal for Narcissus, as they aim to support non-deterministic formats like Protobuf.

Fast Execution & Verification. Only Vest simultaneously provides fast implementations and fast verification.

Advanced Security. Only EverParse3D guarantees double-fetch freedom (useful in concurrent settings). Only Vest provides basic digital side-channel resistance.

9 Discussion and Future Work

Security Lessons. Parsers and serializers have historically been a major source of security vulnerabilities. Vest for the first time demonstrates that even *without formal methods expertise*, programmers can *productively* develop parsers and serializers that are *provably secure* (including side-channel resistance), *fast*, and *easy to use*. Vest achieves these conflicting goals by implementing ergonomic, performant, zero-copy parsers and in-place serializers in Rust; formally verifying the security properties thereof with Verus; efficiently composing the implementations and proofs via trait-based combinators; and exposing the combinators through VestDSL, a DSL that is easy for non-experts to use.

Verification Lessons. During the development of Vest, we learned that formal verification is not just a tool for giving provable correctness/security guarantees to programs, but also a powerful tool for *designing* high-quality software. For example, during the specification and verification of the `Opt` and `Choice` combinators, we discovered that the roundtrip theorems could not be proved without having restrictions on the combinators' arguments, which led us to formally define the concepts of *disjointness* and *productivity* (§4.1). Likewise, when proving the security properties of the `AndThen(A, B)` combinator, we found that `A` and `B` must always consume exactly the same number of bytes (§3.1). In contrast, unverified frameworks [3] often have no such restrictions on combinators, leading to potential ambiguities and surprising behaviors that can only be discovered at runtime.

Additionally, because our trait-based interface (§3.2) enabled quick iteration of the "specify-implement-verify" workflow, we were able to efficiently develop and verify *optimizations* for the combinators. For example, by enhancing the pre- and post-conditions of parsing and serialization, we were able to provably eliminate most runtime checks (e.g., bound checks, checks for arithmetic overflows, termination checks, etc.), leading to cleaner implementations and more performant code than unverified frameworks [3].

Future Work. While Vest is strictly more expressive than regular grammars (with context sensitivity/lookahead introduced by dependent and variant combinators) and can express a wide variety of real-world complex formats, it cannot cover arbitrary formats (which require a Turing Machine). Adding support for recursive definitions [20] would bring Vest's expressiveness on par with PEG/CFG-based parser generators, though recursive definitions are rarely found in security-critical formats (as they may give attackers control over the amount of stack used). Other future work includes extending Vest to support *bit-precise* formats [20, 33], which would permit more compact and efficient parsers and serializers; supporting *streaming* formats, which would allow for incremental parsing of large data streams [34]; and finding ways to securely support *malleable* formats [58].

Ethical Considerations

This research investigates using formal verification to make parsers and serializers more correct and secure, and thus does not carry significant ethical concerns. No attacks or vulnerabilities in public software were discovered as a result of this work.

Open-Science Policy

Our implementation of Vest (§5) and the accompanying case studies (§6) are publicly available in our open-source repository at `https://github.com/secure-foundations/vest`. Additionally, a permalink to Vest's artifact can be accessed at `https://doi.org/10.5281/zenodo.15611769`.

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 their helpful comments and suggestions. We thank Chris Hawblitzel and the rest of the Verus team for their constant support and feedback on Verus-related issues, especially on the development of VestLib. We also thank Nikhil Swamy and Tahina Ramananandro helping us understand the properties of EverParse and EverParse3D.

References

[1] Construct 2.10. https://github.com/construct/construct.

[2] Hammer. https://github.com/abiggerhammer/hammer.

[3] nom, eating data byte by byte. https://github.com/rust-bakery/nom.

[4] wasmbin. https://github.com/RReverser/wasmbin.

[5] What is BinData? https://github.com/dmendel/bindata.

[6] CVE-2021-24119 (mbedTLS base64 PEM side-channel vulnerability). https://nvd.nist.gov/vuln/detail/CVE-2021-24119, July 2021.

[7] Cranelift. https://cranelift.dev/, 2024.

[8] Criterion.rs. https://github.com/bheisler/criterion.rs, 2024.

[9] Polybench/c. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1, 2024.

[10] Rust bitcoin. https://github.com/rust-bitcoin/rust-bitcoin, 2024.

[11] Rustls. https://github.com/rustls/rustls, 2024.

[12] wasmparser. https://github.com/bytecodealliance/wasmparser, 2024.

[13] WebAssembly binary format. https://webassembly.github.io/spec/core/binary/, 2024.

[14] D. E. E. 3rd. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, Jan. 2011.

[15] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving c compiler. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[16] K. Bhargavan, A. Delignat-Lavaud, A. Pironti, A. Langley, and M. Ray. Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension. RFC 7627, Sept. 2015.

[17] Bitcoin Wiki. Protocol documentation: Variable length integer. https://en.bitcoin.it/wiki/Protocol_documentation#Variable_length_integer. Accessed January, 2025.

[18] Bitcoin Wiki. Transaction. https://en.bitcoin.it/wiki/Transaction Accessed January, 2025.

[19] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of nesl. *ACM SIGPLAN Notices*, 31(6):213–225, 1996.

[20] C. Bormann and P. Hoffman. Concise binary object representation (cbor). RFC 8949, Dec. 2020.

[21] J. Bosamiya, W. S. Lim, and B. Parno. Provably-safe multilingual software sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*, August 2022.

[22] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda — a functional language with dependent types. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics*, 2009.

[23] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 69–76. IEEE, 2017.

[24] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, 2004.

[25] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, 2019.

[26] A. Delignat-Lavaud. RSA signature forgery attack in NSS due to incorrect parsing of ASN.1 encoded DigestInfo. MITRE CVE-2014-1569, Sept. 2014.

[27] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman. The matter of Heartbleed. In *Proceedings of the Conference on Internet Measurement Conference (IMC)*, 2014.

[28] R. Frost and J. Launchbury. Constructing natural language interpreters in a lazy functional language. *Comput. J.*, 32(2):108–121, Apr. 1989.

[29] D. K. Gillmor. Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS). RFC 7919, Aug. 2016.

[30] Google. Wuffs. https://github.com/google/wuffs.

[31] P. Gutmann. Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7366, Sept. 2014.

[32] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to speed with WebAssembly. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[33] International Telecommunications Union. Information technology - ASN.1 encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER). https://www.itu.int/rec/T-REC-X.690/en. Accessed May, 2025.

[34] ISO/TC 171/SC2. ISO 32000-2:2020 (PDF 2.0). International Standard ISO 32000-2:2020, International Organization for Standardization, 2020.

[35] S. Klabnik and C. Nichols. *The Rust Programming Language*. No Starch Press, USA, 2018.

[36] A. Langley. A Transport Layer Security (TLS) ClientHello Padding Extension. Internet-Draft draft-ietf-tls-rfc8446bis-11, Internet Engineering Task Force, Oct. 2015.

[37] 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 (SOSP)*, November 2024.

[38] 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. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2023.

[39] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, June 2013.

[40] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, Feb. 2019.

[41] Z. Lin, M. McLoughlin, P. Singh, R. Brennan-Jones, P. Hitchcox, J. Gancher, and B. Parno. Towards practical, end-to-end formally verified x.509 certificate validators with Verdict. In *Proceedings of the USENIX Security Symposium*, Aug. 2025.

[42] E. Lombrozo, J. Lau, and P. Wuille. Bip 141: Segregated witness (consensus layer). https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki, 12 2015.

[43] N. D. Matsakis and F. S. Klock. The Rust language. *Ada Lett.*, 34(3):103–104, Oct. 2014.

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

[45] D. McGrew and E. Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764, May 2010.

[46] Mitre. 2024 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2024/2024_cwe_top25.html Accessed January, 2025., Nov. 2024.

[47] B. Moeller, N. Bolyard, V. Gupta, S. Blake-Wilson, and C. Hawk. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492, May 2006.

[48] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.

[49] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in F*. *PACMPL*, 1(ICFP), Sept. 2017.

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

[51] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. Internet-Draft draft-ietf-tls-rfc8446bis-11, Internet Engineering Task Force, Sept. 2024. Work in Progress.

[52] P. Singh, J. Gancher, and B. Parno. OwlC: Compiling security protocols to verified, secure, high-performance libraries. In *Proceedings of the USENIX Security Symposium*, Aug. 2025.

[53] 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 POPL*, 2016.

[54] 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 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 31–45, 2022.

[55] The Coq Development Team. The Coq Proof Assistant Reference Manual, version 8.7, Oct. 2017.

[56] F. Tramèr, D. Boneh, and K. G. Paterson. Remote side-channel attacks on anonymous transactions. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[57] M. van Geest and W. Swierstra. Generic packet descriptions: verified parsing and pretty printing of low-level data. In *Proceedings of the 2nd ACM SIGPLAN Inter-*

national Workshop on Type-Driven Development, TyDe 2017, page 30–40, New York, NY, USA, 2017. Association for Computing Machinery.

[58] K. Varda. Protocol buffers. https://developers.google.com/protocol-buffers/, 2008.

[59] T. Wallez, J. Protzenko, and K. Bhargavan. Comparse: Provably secure formats for cryptographic protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 564–578, New York, NY, USA, 2023. Association for Computing Machinery.

[60] P. Wouters, H. Tschofenig, J. I. Gilmore, S. Weiler, and T. Kivinen. Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7250, June 2014.

[61] P. Wuille. BIP62: Dealing with malleability. https://bips.dev/62/, Mar. 2014.

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

A Detailed Trait Definitions

In real-world systems, binary format parsers and serializers are performance-critical components that need to be implemented in a low-level, efficient systems programming language. For VestLib, we use Rust due to its combination of high-level abstractions, low-level control, and strong safety guarantees.

Figure 12 shows the detailed definitions for our three main traits. Both the `Combinator` and `SpecCombinator` traits have an associated type `Type` that represents the internal data type of a format and a pair of `parse` and `serialize` functions. (We abbreviate specification functions with `s_`). The associated type for a `Combinator` would be the actual data type that the combinator parses and serializes (like a `&[u8]` for all bytes combinators or a `Vec<T>` for all repetition combinators), while the corresponding associated type for a `SpecCombinator` would be the mathematical view of that data type (like a `Seq<u8>` for all bytes combinators and `Seq<T>` for all repetition combinators). The `SecureCombinator` trait extends the `SpecCombinator` trait with a set of formally proven properties that ensure the correctness and security of the combinator's trusted specification. As discussed in §3.2, the trait bound `Self: View` and `Self::V: SecureCombinator<Type = <Self::Type as View>::V>` for the `Combinator` trait ensures that every combinator has a view, and that view should be a `SecureCombinator` with the same internal data type (modulo the view).

Notably, the `Combinator` trait is generic over two types `I` and `O`, which represent the input and output buffer types, respectively. `I` and `O` are further given trait bounds `VestInput` and `VestOutput`; the definitions for these traits are shown in Figure 13. The `VestInput` and `VestOutput` traits define a common interface for input and output buffers. Rust's built-in buffer types (like `&[u8]`, `Rc<Vec<u8>>`, etc.) or custom buffer types can implement these traits. VestLib users can, for example, instantiate all input and output buffer types in `Combinator<I, O>` with `&[u8]` and `Vec<u8>`. As discussed in §4.2, `VestInput` and `VestOutput` enable our side-channel security guarantee via type abstraction. We also have subtraits `VestPublicInput` and `VestPublicOutput`, which extend `VestInput` and `VestOutput` to allow fully public access to the bytes; these are used for combinators that must view or manipulate bytes, such as integers.

As an illustration, we now discuss `parse` and `serialize`, specialized to the common use case where we use `VestPublicInput`/`VestPublicOutput` on public bytes. After specializing, we receive these signatures:

- `parse: &[u8] → PResult<T>`
- `serialize: (&T, &mut Vec<u8>, usize) → SResult<usize>`

Here, `T` represents the internal data type of the format. The `parse` function takes a byte slice and returns a `Result` containing either a tuple of the number of bytes consumed and the parsed value of type `T`, or an error. The `serialize` function takes an immutable reference to the value of type `T`, a mutable reference to a `Vec` for the destination byte vector, an offset in the byte vector, and returns a `Result` with the number of bytes written to the byte vector or an error. The `Result` type is a sum type with `Ok` and `Err` variants, indicating the success or failure of the parsing and serialization operations.

Importantly, the `parse` function operates on an input buffer of a reference type (`&[u8]`), and the parsed values are often *borrowed forms* of the input buffer. This design ensures that parsing can be *zero-copy* and *non-allocating* whenever possible, meaning that it parses the input buffer without duplicating its contents or allocating new memory. The only exception is when the format requires a dynamically-known number of repetitions, in which case the parser needs to allocate a vector to store the parsed values (the elements of the vector would still be borrowed forms of the input buffer). On the other hand, the `serialize` function operates on a pre-allocated output buffer of a vector and writes the serialized bytes to the buffer in-place, avoiding the overhead of heap allocation during serialization.

```
type PResult<Type> = Result<(usize, Type), Error>;
type SResult<Type> = Result<Type, Error>;

/// The specification (view) of an implementation combinator.
trait SpecCombinator {
/// The view of [Combinator::Result].
type Type;

spec fn s_parse(self, s: Seq<u8>) → PResult<Self::Type>;
spec fn s_serialize(self, v: Self::Type) → Seq<u8>;
}

/// The properties of a secure combinator.
trait SecureCombinator: SpecCombinator {
proof fn theorem_serialize_parse_rt(&self, v: Self::Type)
  ensures
      self.s_parse(self.s_serialize(v)) ==
        Ok((self.s_serialize(v).len(), v))
;
proof fn theorem_parse_serialize_rt(&self, b: Seq<u8>)
  ensures
    self.s_parse(b) matches Ok((n, v)) ⟹
      self.s_serialize(v) == b[..n]
;
// More lemmas and corollaries...
}

/// Implementation for parser and serializer combinators.
trait Combinator<I, O> where Self: View,
  I: VestInput, O: VestOutput<I>,
  // A combinator's view must be a [SecureCombinator].
  Self::V: SecureCombinator<Type = <Self::Type as View>::V>,
{
/// The output type of parsing.
type Type: View;

/// The input type of serialization.
type SType: View<V = Self::Type::V>;

exec fn parse(&self, input: I) → (r: PResult<Self::Type>)
  ensures
    r matches Ok((n, v)) ⟹
      self@.s_parse(input@) == Ok((n, v@)),
    r is Err ⟹ self@.s_parse(input@) is Err
;

exec fn serialize(&self, v: Self::Type, buf: &mut O,
            pos: usize) → (r: SResult<usize>)
  ensures
    buf@.len() == old(buf)@.len(),
    r matches Ok(n) ⟹ self@.s_serialize(v@).len() == n &&
      buf@ == splice(old(buf)@, pos, self@.s_serialize(v@))
;
}
```

Figure 12: The definitions for the `Combinator`, `SpecCombinator`, and `SecureCombinator` traits in VestLib (simplified). `matches` is a Verus keyword in `spec`-mode that checks if a pattern matches a value. `@` is a sugar for `.view()`. `splice` is a `spec`-mode function that inserts a byte sequence into another byte sequence at a given position.

```
trait VestInput: {
  fn len(&self) → (res: usize);
  fn subrange(&self, i: usize, j: usize) → (res: Self);
}

trait VestOutput<I> {
  fn len(&self) → (res: usize);
  fn set_range(&mut self, i: usize, input: &I);
}

trait VestPublicInput: VestInput {
  fn as_byte_slice(&self) → (res: &[u8]);
}

trait VestPublicOutput<I>: VestOutput<I> {
  fn set_byte_range(&mut self, i: usize, input: &[u8]);
}

/// Implementations for common buffer types
impl<'a> VestInput for &'a [u8] {
  fn len(&self) → usize {
    <[u8]>::len(self)
  }
  fn subrange(&self, i: usize, j: usize) → &'a [u8] {
    slice_subrange( *self, i, j) /// Verus primitive
  }
}
impl<'a> VestPublicInput for &'a [u8] {
  fn as_byte_slice(&self) → &[u8] {
    *self
  }
}
impl<I: VestPublicInput> VestOutput<I> for Vec<u8> {
  fn len(&self) → usize {
    Vec::len(self)
  }
  fn set_range(&mut self, i: usize, input: &I) {
    /// Verus primitive on slices
    set_range(self, i, input.as_byte_slice());
  }
}
impl<I: VestPublicInput> VestPublicOutput<I> for Vec<u8> {
 fn set_byte(&mut self, i: usize, value: u8) {
    self.set(i, value); /// Equivalent to self[i] = value
  }
  fn set_byte_range(&mut self, i: usize, input: &[u8]) {
    /// Derived function, loops over indices
    set_range(self, i, input);
  }
}
```

Figure 13: **Definitions of the `VestInput`, `VestOutput`, `VestPublicInput`, and `VestPublicOutput` traits**. *Details, including Verus specifications that enforce correct implementations, are elided.*