Secure Parsing and Serializing with Separation Logic Applied to CBOR, CDDL, and COSE

TAHINA RAMANANANDRO, GABRIEL EBNER, GUIDO MARTÍNEZ, and NIKHIL SWAMY, Microsoft Research, USA

Incorrect handling of security-critical data formats, particularly in low-level languages, are the root cause of many security vulnerabilities. Provably correct parsing and serialization tools that target languages like C can help. Towards this end, we present PulseParse, a library of verified parser and serializer combinators for non-malleable binary formats. Specifications and proofs in PulseParse are in separation logic, offering a more abstract and compositional interface, with full support for data validation, parsing, and serialization. PulseParse also supports a class of recursive formats—with a focus on security and handling adversarial inputs, we show how to parse such formats with only a constant amount of stack space.

We use PulseParse at scale by providing the first formalization of CBOR, a recursive, binary data format standard, with growing adoption in various industrial standards. We prove that the deterministic fragment of CBOR is non-malleable and provide EverCBOR, a verified library in both C and Rust to validate, parse, and serialize CBOR objects implemented using PulseParse. Next, we provide the first formalization of CDDL, a schema definition language for CBOR. We identify well-formedness conditions on CDDL definitions that ensure that they yield unambiguous, non-malleable formats, and implement EverCDDL, a tool that checks that a CDDL definition is well-formed, and then produces verified parsers and serializers for it.

To evaluate our work, we use EverCDDL to generate verified parsers and serializers for various security-critical applications. Notably, we build a formally verified implementation of COSE signing, a standard for cryptographically signed objects. We also use our toolchain to generate verified code for other standards specified in CDDL, including DICE Protection Environment, a secure boot protocol standard. We conclude that PulseParse offers a powerful new foundation on which to build verified, secure data formatting tools for a range of applications.

CCS Concepts: • **Software and its engineering** → *Source code generation*; **Specification languages**; **Correctness**; **Formal software verification**; • **Information systems** → **Data layout**; **Data encoding and canonicalization**; • **Security and privacy** → *Key management*; *Embedded systems security*; **Trusted computing**; **Management and querying of encrypted data**.

Additional Key Words and Phrases: Binary data formats, CBOR, CDDL, COSE, DICE, DPE, Formal verification, Measured boot, Secrets management

1 Introduction

Incorrect handling of security-critical data formats, be it in parsing attacker controlled data, or in serializing data for cryptographic applications, is a major source of security vulnerabilities [Finney 2006; MITRE 2016]. In response, there is a rich area of research into tools for secure parsing [Bangert and Zeldovich 2015; Diatchki et al. 2024; Lasser et al. 2021; Mundkur et al. 2020; Ramananandro et al. 2019]. We are particularly interested in secure handling of binary data formats for use in security-critical low-level applications, in C and other systems programming languages, including in OS components, embedded systems, and in cryptographic applications.

Authors' Contact Information: Tahina Ramananandro, taramana@microsoft.com; Gabriel Ebner, gabrielebner@microsoft.com; Guido Martínez, guimartinez@microsoft.com; Nikhil Swamy, nswamy@microsoft.com, Microsoft Research, Redmond, WA, USA.

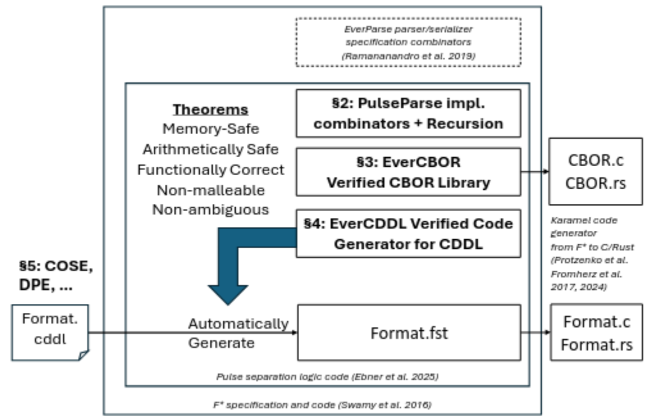

Fig. 1. Architecture of our contributions

In this context, formally proven parser generators have been used to secure critical, commercial software including in Microsoft's OS and cloud infrastructure [Swamy et al. 2022]. However, such uses have focused primarily on validating flat, tag-length-value encodings of network packet formats. We aim to broaden the scope of secure, low-level binary formatting tools, enabling them to handle richer formats (such as those with certain forms of recursion) and to flexibly support both parsing and serialization, in a performant, zero-copy-by-default, low-level style.

Our *first contribution* (§2) is PulseParse, a new verified library for secure parsing and serialization. PulseParse is implemented in F* [Swamy et al. 2016] and in its separation logic sub-language Pulse [Ebner et al. 2025], with formal proofs of memory safety, functional correctness, and non-malleability (i.e., unique binary representation) of formats. The design of PulseParse employs a novel application of separation logic to parser & serializer combinators, yielding an abstract style of specification with compositional proofs. PulseParse also supports a class of security-relevant recursive formats, namely those that can be validated in constant stack space.

Recursion in PulseParse is essential to model CBOR (Concise Binary Object Representation) [Bormann and Hoffman 2020], an Internet standard for the binary representation of general-purpose JSON-like data structures. A subset, Deterministically Encoded CBOR, aims to offer non-malleability, thus avoiding hashing-based authentication bugs that have occurred in similar binary formats [Decker and Wattenhofer 2014]. Our *second contribution* (§3) is EverCBOR, a formalization of CBOR, including a proof that its deterministic encoding is indeed non-malleable—the first such proof. Implemented in PulseParse, EverCBOR produces verified code in both C and safe Rust for validating, parsing, and serializing CBOR objects.

CBOR is a single uniform data format, deferring data specifications to a high-level schema language for CBOR items, called

CDDL (Concise Data Definition Language) [Birkholz et al. 2019]. By splitting those two concerns, CBOR and CDDL greatly help protocol designers specify data schemas with extensibility and forward-compatibility in mind. Our *third contribution* (§4) is a formalization of CDDL, including inferring well-formedness conditions on CDDL definitions (in the form of a new elaboration algorithm, proven sound) that yield unambiguous and non-malleable formats. Our formalization takes the form of a tool, EverCDDL, that first formally proves that a CDDL definition is well-formed, and then generates a custom data type along with corresponding low-level parsers and serializers in PulseParse, formally verified for inverse, non-malleability, memory safety, and functional correctness with respect to the CBOR and CDDL specifications.

CDDL is used in dozens of other standards, in applications including supply-chain integrity (SCITT) [Birkholz et al. 2025], device attestation protocols (DPE) [Trusted Computing Group 2023], and WebAuthn passwordless authentication [Word Wide Web Consortium 2019]. Perhaps its most prominent use is in the specification of COSE (CBOR Object Signing and Encryption) [Schaad 2022], a standard for cryptographically signed and encrypted objects, certificates, and keys, itself used in security-critical applications such as SCITT, DPE, Client-to-Authenticator Protocol (CTAP) [Büttner and Gruschka 2023; The FIDO Alliance 2025], and vaccine certificates [European Union eHealth Network 2021]. Our *fourth contribution* (§5) is to evaluate our libraries on two applications, COSE and DPE. First, we show how to adapt the CDDL specifications of COSE and DPE so that they are provably unambiguous and non-malleable (using EverCDDL), and then integrate the resulting parsers and serializers in verified applications. For COSE, we produce a verified library for COSE signing, relying on verified cryptographic implementations from HACL★ [Zinzindohoué et al. 2017], proving that the payload of a signature object is exactly the signature of the to-be-signed bytes with the given key. For DPE, we show how to integrate our verified parsers and serializers with a prior verified implementation [Ebner et al. 2025].

Figure 1 shows the overall architecture of our contributions. All the theorems in this paper and the software described are formally verified in F★ [Swamy et al. 2016], with the separation logic parts developed in Pulse [Ebner et al. 2025], an embedded language in F★. All our code is proven memory safe, arithmetically safe, and functionally correct, and the formats we formalize are all proven non-malleable. Our software can be used in verified Pulse applications, as we do for COSE and DPE. Additionally, using Karamel, an existing code generator [Fromherz and Protzenko 2024; Protzenko et al. 2017], EverCBOR extracts from Pulse to a standalone library in C and in safe Rust, with idiomatic, defensive APIs as a drop-in replacement of existing unverified CBOR libraries used in a variety of applications. We evaluate EverCBOR against commonly used unverified CBOR libraries such as QCBOR [Lundblade 2023] and TinyCBOR [Intel 2021], noting that we support more CBOR features (including arbitrary maps), and implement all necessary checks, find that our verified code is competitive in speed and memory consumption. Verified code produced by EverCDDL also extracts to a standalone library in either C or safe Rust.

All the artifacts we contribute have been merged into EverParse[1]. Note, throughout the paper, we say "format" to mean "parsing and serialization", e.g., we say "format combinators" to mean "parser and serializer combinators".

2 PulseParse: Format Combinators with Separation Logic

Combinator parsing has its roots in functional programming [Hutton 1989], providing a higher-order, compositional way to structure parsers. We seek to use combinator parsing to produce verified code in low-level languages, a technique leveraged first by EverParse [Ramananandro et al. 2019], a format combinator library in F★ with formal proofs of correctness and security, yielding verified C code. Their verification approach is to layer the combinators, distinguishing between *specification* combinators, pure functions that *define* the data format specification, and on which proofs of properties such as non-malleability are conducted; and *implementation* combinators which follow the structure of the specification combinators while refining them to efficient, low-level code.

PulseParse follows this approach too. In fact, we simply reuse many of the specification combinators from EverParse, though we contribute some new specification combinators, notably for recursive formats. Our first main innovation is a new library of implementation combinators, whose proofs are structured using separation logic, contrary to EverParse, which uses a classical Hoare logic. As we will explain, through the use of separation logic, PulseParse proofs are more modular, and enable abstract implementation combinators, simplifying both their construction and, more importantly, proofs of their clients.

2.1 Specification Combinators (Review)

A *parser specification* in PulseParse is a pure F★ function of type `parser t = seq U8.t → option (t & nat)` (with an extra non-malleability condition as below), that takes as argument a sequence of input bytes, and returns `Some(v, n)` if parsing succeeds and the first `n` input bytes are a binary representation of the high-level value `v`; and `None` if parsing fails.

Such a parser specification defines the data format, and properties about the format can be proven as lemmas. EverParse parser specifications are required to be *non-malleable*: for a given data format defined by its parser specification `p`, if `p(b1) = Some (v, n1)` and `p(b2) = Some (v, n2)`, parsing two input byte sequences `b1` and `b2` to the same high-level value `v`, then `n1 = n2` and `b1` and `b2` coincide on their first `n1` bytes, meaning that the first `n1` bytes of `b1` are a unique binary representation for `v`. Non-malleability is especially important for security-critical applications, especially in cryptographic contexts—several prominent attacks come down to malleability of formats [Finney 2006; MITRE 2016].

To combine parsers, e.g., `p1: parser t1` and `p2: parser t2`, one uses a combinator `parse_pair p1 p2: parser (t1 & t2)`, a parser specification for a pair of values whose binary representations are laid out side-by-side. Since `p1` and `p2` are non-malleable, one can prove that `parse_pair p1 p2` is also non-malleable by construction.

Given a parser specification `p: parser t`, a serializer specification for `p` has type `serializer p = (x: t) → (b: seq U8.t { p b == Some (v, length b) })`:

[1] https://github.com/project-everest/everparse/

serializing a high-level value (v) yields a sequence of bytes guaranteed to parse back to v, specified with a *refinement type* on the return value. Serializers can also be combined, e.g., given s1: serializer p1 and s2: serializer p2, the pair serializer serialize_pair s1 s2 serializes a (v1, v2): t1 & t2, by running s1 on v1, then s2 on v2, and concatenate the resulting byte sequences. The combinator serialize_pair s1 s2 has type serializer (parse_pair p1 p2) only if p1 has the *prefix property*: for any sequence of input bytes b such that p(b) returns Some(v, n), p(b') returns the same result for any input b' coinciding with b on its first n bytes. This property is necessary to prove the correctness of serialize_pair with respect to parse_pair.

PulseParse reused EverParse's specification combinators for various types, such as machine integers, bit fields, value-dependent pairs, lists of a given number of elements, checking for a value property, and rewriting under a bijection.

2.2 Implementation Combinators with Separation Logic

For implementation combinators, PulseParse uses *separation logic* [Reynolds 2002], as provided by Pulse [Ebner et al. 2025]. For a given pair of parser and serializer specification combinators, we implement several combinators in Pulse: *validators*, *jumpers*, *accessors* and *readers* for parsing; and *writers* for serialization.

Validators For p:parser t and s:serializer p, a *validator* v is a Pulse function (i.e., a procedure with possible side effects) that takes an input byte array and returns the number of bytes consumed by p if parsing succeeds, or an error code if parsing fails. Whereas p is specified on an input sequence of bytes, v reads the contents of a concrete array stored in memory, recording the number of bytes read in a mutable out parameter storing a machine integer U64.t, and returning true if, and only if, p would have succeeded on the abstract byte contents of the array. Validators can also be combined (e.g., validate_pair v1 v2). F* inlines the combinator definition when transpiling to C or Rust, so that the resulting code is first-order.

Jumpers A *jumper* is a Pulse function that takes an input byte array required to start with a valid byte representation with respect to p, and returns the number of bytes consumed—it is used to "jump" over a known valid item in a byte array.

Accessors An *accessor* is a Pulse function that takes an input byte array containing a valid byte representation, and returns a (pointer to a) subarray containing the valid byte representation of a subobject. PulseParse is careful to ensure that no heap accesses are incurred in the process. Accessors specified in separation logic can be pleasingly abstract. Consider for instance the parse_pair combinator for parsing a pair of data. For p1:parser t1, p2:parser t2, s1:serializer p1 and s2:serializer p2, to implement a pair accessor, we need the jumper j1 for p1, to jump over the first component of the pair. Then, for a byte array a and a pair (v1:t1, v2:t2), we introduce a separation logic predicate, ser (serialize_pair s1 s2) a (v1, v2), stating that the current contents of the byte array a is exactly the byte representation of (v1, v2) obtained by the corresponding pair serializer specification. Then, we specify a call to a pair accessor implementation in Pulse using the following separation logic triple—this is our first glimpse of separation logic and we explain the specification in detail below.

```
{ ser (serialize_pair s1 s2) a (v1, v2) }
 let (a1, a2) = access_pair j1 a
{ (ser s1 a1 v1 ∗ ser s2 a2 v2) ∗
 ((ser s1 a1 v1 ∗ ser s2 a2 v2) −∗ ser (serialize_pair s1 s2) a (v1, v2)) }
```

The first part of the triple is the *precondition*, which describes the relevant part of memory as a separation logic proposition (of type slprop) required to hold before running the Pulse statement mentioned in the middle part of the triple. In this case, the precondition simply states that before running the accessor, one must prove that the array a contains a valid serialization of (v1, v2).

The last part of the triple is the *postcondition*, describing a property of the memory upon completion of the Pulse statement. The postcondition uses two separation logic connectives. First, $A * B$ is a *separating conjunction*, meaning A and B are separation logic predicates that describing disjoint ownership of memory. Owning the predicate A means no other part of the program can disturb the validity of A. The postcondition also uses a *magic wand*[2], $A \mathrel{-\!*} B$, which is a connective that enjoys an *elimination proof rule* $\frac{A * (A \mathrel{-\!*} B)}{B}$. That is, one can trade an A and (separately) $A \mathrel{-\!*} B$ for a B.

Specifically, in the context of the triple above, the postcondition says that (1) a1 points to an array segment that contains a valid serialization of v1; (2) a2 points to an array segment that contains a valid serialization of v2; and (3) one can give up ownership of a1 and a2 to recover ownership of the entire original array segment a that contains a serialization of (v1, v2). This specification captures the essence of *zero-copy* parsing with no heap allocation—the caller gains ownership to the relevant array segments, and when it is done with them, it can simply relinquish ownership and use the magic wand to regain ownership to the original array a.

Note how this specification makes no mention of array offsets, which are abstracted away. Using a similar pattern, we provide an accessor for dependent pairs, accessors for the head and the tail of a nonempty list of elements. The pattern $A * (A \mathrel{-\!*} B)$ is common enough that we abbreviate it as $A \mathrel{>\!\!*} B$.

Readers For base values such as integers, we provide PulseParse *readers*, to return the actual values. Given a parser-serializer specification pair p: parser t, s: serializer p, a reader applied to an array a satisfies the following triple:

```
{ ser s a v } let v' = reader s a { ser s a v ∗ (v' == v) }
```

showing that it returns a value v' equal to the value v from the precondition, without changing the array a.

Now, for r1: reader s1, j:jumper p1, and r2: reader s2, we can implement read_pair r1 j1 r2 : reader (serialize_pair s1 s2):

```
let (x1, x2) = access_pair j1 x;
let v1 = r1 x1; let v2 = r2 x2;
wand_elim _ _; (∗ a ghost proof step for the elimination rule for −∗ ∗)
(v1, v2)
```

Unlike in EverParse, PulseParse requires no offset reasoning.

In PulseParse, we also provide value readers for dependent pairs and bitfields. However, we do not provide value readers for lists or other variable-sized data, since we want to parse data without heap

[2] Our proofs use Pulse's *trades*, which have a slightly different semantic model than magic wands, but we only rely on usual proof rules that are valid for both the traditional magic wand and Pulse trades.

allocating, and only using constant stack space with respect to the input, which may be attacker-controlled.

Rather, we provide *zero-copy readers*, which are functions that "save" the pointers to subcomponents without parsing them. To this end, for a p: parser t, and s: serializer p, one needs to choose a low-level datatype representation u into which to parse such data, along with a separation logic predicate r: u → t → slprop. Then, a zero-copy reader is a Pulse function of the following signature:

{ ser s a v } **let** res = zerocopy s r a { r res v >∗ ser s a v }

Similarly to accessors, the postcondition uses the *magic wand*, thus allowing to "borrow" permissions from subpointers of x into res, controlled by r. For instance, a zero-copy reader can read a pair of a machine integer and a sequence of bytes, by returning the value of the integer and a pointer to the byte array, which it will thus not copy. Then, the application can use further accessors and readers to manipulate that byte array. We provide several zero-copy reader combinators, e.g., a value reader and a value-dependent zero-copy reader can be combined together to obtain a zero-copy reader for a dependent pair of values. In PulseParse, we provide the following zero-copy reader combinators:

- A value reader can be lifted as a zero-copy reader, by setting the low-level type u to be identical to the high-level type t, and r x y = pure (x == y).
- A byte array valid with respect to some serializer specification s can be left as is: u = byte_array and r x y = ser s x y.
- Two zero-copy readers can be paired together to obtain a zero-copy reader for a pair of values.
- A value reader and a value-dependent zero-copy reader can be combined together to obtain a zero-copy reader for a dependent pair of values.
- The high-level values read by a zero-copy reader can be rewritten with a bijection without changing the low-level value returned by the zero-copy reader.
- A zero-copy reader for low-level type u1 and relation r1 can be turned into a zero-copy reader for low-level type u2 and relation r2 and the same high-level values, if provided a Pulse function going from u1 to u2:

  (x1: u1) → (y: t) → stt u2 (**requires** r1 x1 y)
  (**ensures** λ (x2: u2) → r2 x2 y ∗∗ (r2 x2 y −∗ r1 x1 y))

We provide no zero-copy reader that would allocate into the heap. In particular, we do not allocate references, byte arrays, or other kinds of arrays.

Writers Similarly to value readers, we provide value writer combinators, taking a value to write, and an output byte array, and returning the number of bytes written. We also provide value size combinators to compute the minimal size required for serialization; these value size combinators take a bound as argument, and gracefully fail if the size needed is larger than the bound, so as to avoid arithmetic overflows.

However, contrary to input, an application may want to build data structures for which it entirely controls nesting and memory usage, apart from byte arrays containing unparsed data. For this, we define *copy writers* for generic data structures so that an application can define such a data structure, populate it in any order, and serialize it all at once by copying it into an output buffer following a serializer specification. Then, we specify copy writers as:

{ ∃ w. out ↦ w ∗ r vl vh ∗ (length w ≤ length (s vh)) }
let res : size_t = writer s r a out
{ ∃ w'. out ↦ (append (s vh) w') ∗ r vl vh }

where s is a serializer specification, r : tl → th → slprop is a relation between the actual data structure vl:tl that the application built and wants to serialize and vh:th, some abstract specification-level value. A writer expects as its precondition that the application has proven vl and vh are related, and that the output buffer is large enough to contain the serialized bytes s(y).

The separation logic predicate r plays a critical role in the correctness of the PulseParse copy writers. For instance, tl can be the Pulse type ref th of non-null pointers to values of type th, in which case r vl vh will be vl ↦ vh, the predicate saying that the contents of the reference vl is vh. Then, that writer will read the contents of vl, which is equal to vh, and then call a value writer. We provide many copy writer combinators, including, for example, given two copy writers for low-level types tl1 and tl2 and two separation logic predicates for the same high-level type th, we provide a copy writer for low-level type tl1 + tl2, thus allowing several low-level data structures for the same high-level type. Lacking support for abstract relations between low and high-level representations, serialization in EverParse requires applications to directly write into the output buffer in the right order, incurring heavy application-level reasoning about output offsets.

More generally, in PulseParse, we define the following copy writer combinators:

- we lift value writers to copy writers using th = tl and r x y = pure (x == y)
- we provide a byte-copy writer for byte arrays containing unparsed bytes valid with respect to serializer specification s (with tl = byte_array and r x y = ser x y)
- given a copy writer for low-level type tl, we provide a copy writer for low-level type ref tl
- given a copy writer for low-level type tl, we provide a copy writer for low-level type array tl n to serialize an array of *n* elements.
- given two copy writers for low-level types tl1 and tl2, we provide a pair copy writer for low-level type tl1 & tl2
- given two copy writers for low-level types tl1 and tl2 and two separation logic predicates for the same high-level type th, we provide a pair copy writer for low-level type tl1 + tl2, thus allowing several low-level representations for the same high-level type.
- given a value writer for the left-hand-side of a value-dependent pair, and a copy writer for the right-hand-side, we provide a value-dependent pair copy writer

In Section 4, we define a set of CDDL serializer combinators using a similar methodology, which we extend to define parser combinators using low-level representations that can contain both application-controlled data structures and user-controlled unparsed data. Based on our learnings from CDDL, we integrated similar parsing combinators in PulseParse—Appendix A shows them in use on a small recursive format for arithmetic expressions.

Support for some recursion Most memory-constrained binary data parsers do not support arbitrary recursion, because many recursive formats require a stack or other form of memory whose size would grow with the input, thus exposing themselves to attackers exhausting memory during validation or parsing.

However, there is a class of recursive data formats that allow validation in constant stack and memory space—we will see in Section 3 that CBOR belongs to this class.

THEOREM 2.1. *Consider a binary data format where an object representation starts with a header, followed by a contiguous sequence of recursive object payload entries, and nothing afterwards. If a header can be validated in constant stack and memory space, and if the header of an object alone is enough to determine the number of immediate children of this object, then data in this format can be validated in constant stack and memory space.*

To support this class, we include in PulseParse a recursive parser specification combinator parse_rec as follows: let th be the high-level type of headers, t be the high-level type of objects, p a parser specification for headers consuming at least one byte, count a function computing the number of recursive payload elements, and synth a function synthesizing a high-level object from its header and elements; then, parse_rec is defined below, where **let**! sequences option computations:

```
let rec parse_rec' (ph:parser th) (count:th → nat)
 (synth: (h: th) → (l: nlist t (count h)) → t) (fuel: nat) (b: Seq.seq U8.t)
: option (t & nat)
= if fuel = 0 then None else
  let! h, size = ph b in (* parse th header *)
  let b' = slice_from b size in (* b' contains (count h) elements to be parsed *)
  let! l = parse_nlist (parse_rec' ph count synth (fuel−1)) (count h) b' in
  Some (synth h l) (* map the parsed values to a high−level value t *)
let parse_rec ph count synth b = parse_rec' ph count synth (1+length b) b
```

We prove that if a ph header has the prefix property, then so does parse_rec ph count synth, and if synth is injective and ph is non-malleable, then parse_rec ph count synth is non-malleable.

Of course, parse_rec is recursive—but it is only a specification combinator and not meant to be executed. The implementation combinators use only constant stack.[3] For validation, we take as argument a header validator, and a function to retrieve the number of expected payload items in the payload. Then we maintain a counter of expected items, which we initialize to 1. Whenever we start validating an item, we decrease that counter. Then we add the number of expected items in the payload. The validator succeeds if the counter reaches 0. Using Pulse, we prove that our validator is functionally correct with respect to its recursive specification, using a loop invariant. We take care of avoiding arithmetic overflow by leveraging the fact that a valid header always consumes at least one byte. See Appendix B for more details; Appendix A provides an example PulseParse for a recursive format of variable-arity trees.

[3]Note, stack space usage is outside the scope of our formal proof, since the underlying logic does not provide a way to specify it. However, we only use while loops and use no recursive functions, so the function call depth is bounded statically (by the number of function definitions.)

$$
\begin{array}{lcl}
\text{cbor} & ::= & \text{Int } (x \in [-2^{64}; 2^{64}-1]) \\
& | & \text{Simple } (x \in [0,23] \cup [32,255]) \\
& | & \text{ByteString } (n \in [0, 2^{64}-1], x \in [0,255]^n) \\
& | & \text{TextString } (n \in [0, 2^{64}-1], x \in [0,255]^n \cap \text{UTF-8}) \\
& | & \text{Tagged } (\text{tag} \in [0, 2^{64}-1], v : \text{cbor}) \\
& | & \text{Array } (n \in [0, 2^{64}-1], x \in \text{cbor}^n) \\
& | & \text{Map } (n \in [0, 2^{64}-1], x : (\text{cbor} \xrightarrow{n} \text{cbor}))
\end{array}
$$

where $(t \xrightarrow{n} t)$ is the type of extensional maps with n entries

Fig. 2. The data model for CBOR items

We now turn to our formalization of CBOR, making essential use of PulseParse's support for recursion, and abstract separation logic specifications.

3 EverCBOR: A Verified Generic CBOR Parser and Serializer

JSON is a ubiquitous textual representation of data. However, it comes with a vast collection of issues, some related to efficiency (whitespace, decimal integers, etc.), others related to security (parsing errors due to bad nesting of quotes or braces, string injection, etc.) This is why Internet practitioners have long sought binary representation alternatives, such as UBJSON, BSON, or MessagePack.[4] For uniformity and extensibility reasons, the IETF adopted CBOR as an Internet Standard in 2020 as RFC 8949 [Bormann and Hoffman 2020]. Since then, CBOR rapidly evolved into a binary format of its own, defining its own set of items extending JSON.

3.1 Background: CBOR

A CBOR item (Figure 2) can be any one of: a 64-bit nonnegative integer, a 64-bit negative integer represented as "one's complement", a "simple value", a byte string, a UTF-8 text string, a CBOR item tagged with a nonnegative 64-bit integer, a finite array of CBOR items (a heterogeneous ordered sequence,) or a finite key-value map, where each entry *key* or value can be an arbitrary CBOR item—a generalization of JSON, where only strings are allowed as keys. Moreover, simple values are a subset of non-negative byte values, meant to generalize JSON's Boolean type by encoding symbols such as NULL, meant to be distinct from integer values or empty strings.

However, naïvely transcribing the above description as a grammar or an inductive datatype could conflate maps with lists of pairs of items, potentially allowing duplicates in map keys. Thus, secure applications must make sure CBOR maps have no such duplicates, to avoid misunderstandings where different applications will look at different entries for one given key. Moreover, unlike arrays, map entries are unordered. Thus, the entry keys of a map are better modeled as a set rather than as a list. These problems are inherited from JSON. Further, trying to directly define CBOR items as an inductive type in a proof system is not possible, since such a definition would require the type to appear negatively in map keys. So, we look to the byte representation as a basis for formalizing CBOR's data model.

CBOR defines a byte representation for its items in a tag-count-payload fashion. Figure 3 shows how CBOR items are represented as bytes. The first byte contains three bit fields, the most significant

[4]https://ubjson.org/, https://bsonspec.org/, https://msgpack.org/

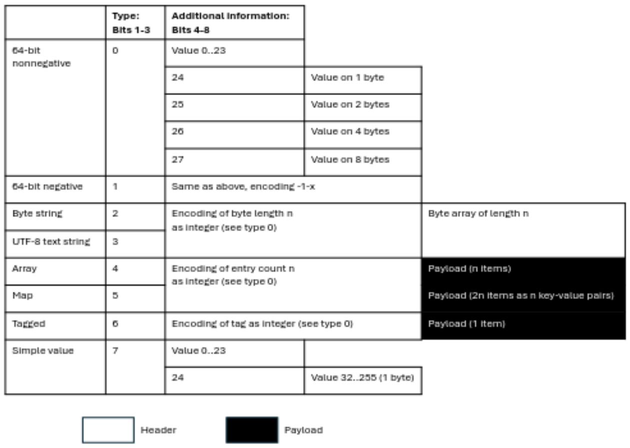

Fig. 3. Representing CBOR items as bytes

3 bits of which describe the type of the CBOR item. The remaining 5 bits, called "additional information", encode an integer from 0 to 31: additional info 0 to 23 encode a nonnegative integer (or a simple value) of this value. For 64-bit integers, info 24, 25, 26 and 27 encode the fact that the integer is encoded in the next 1, 2, 4 and 8 bytes respectively. Thus, integers are encoded in variable length. Byte and text strings are prefixed with their byte size as a 64-bit integer encoded in the same way (starting from the 4th bit of the first byte), thus limiting their byte size to $2^{64} - 1$. Similarly, arrays and maps start with their number of entries as a 64-bit integer (thus limiting their entry count to $2^{64} - 1$,) followed by their entries consecutively, where a map entry consists in two consecutive CBOR items; and tagged items start with their tag as a 64-bit integer, followed by the payload item. Type number 7 is used for simple values.

Not all such binary data represent valid CBOR items. Once binary data conforms with this representation, a validator needs to check for the absence of map key duplicates. We call *raw CBOR bytes* any sequence of bytes conforming to the binary representation but not yet checked for the absence of map key duplicates.

Deterministically-Encoded CBOR A given CBOR item has several possible representations, owing to the variable byte size of integer values and length prefixes, and the order in which map entries are serialized. This allows for *malleability attacks:* if an application cryptographically signs the byte representation of a CBOR item, an attacker could possibly construct a *different* representation of the same item, which the application would not recognize as having signed. This can lead to serious security issues, see e.g. [Decker and Wattenhofer 2014]. To prevent this issue, Deterministically Encoded CBOR [Bormann and Hoffman 2020, §4.2.1], mandates integers to be serialized in their shortest form, and map entries to be serialized in the increasing lexicographic order of the byte representations of their keys. We prove that this subset indeed provides a unique binary representation for all CBOR items. Deterministically Encoded CBOR is used in COSE and many other security-critical protocols requiring unique binary representation.

Indefinite-length CBOR byte representations CBOR can also represent arrays and maps in an indefinite-length way, where, instead of storing the number of entries in the prefix of their byte representation, indefinitely many entries can be parsed until a special "stop" item is encountered. Such representations are explicitly excluded from the deterministic subset of CBOR, which requires all arrays and maps to have definite lengths specified in their byte representation prefixes.

Even beyond the deterministic subset, we believe that indefinite-length representations introduce several security issues. Such representations are mostly meant for processing of streamed input data. Implementations could start processing such streamed data before having reached its end, and thus, an attacker could induce unexpected behavior by providing a CBOR data stream that starts with valid input and suddenly becomes invalid. This is why we choose to validate all bytes at once after making sure we receive it all, and start processing them only once we make sure the whole representation is valid.

With our choice, a CBOR byte representation validator accepting indefinite-length maps and arrays arbitrarily nested with definite-length maps and arrays needs to store the number of CBOR items left to be validated after each "stop" item. For a given n, validating any n-depth nesting of a definite-length array of size at least 2 containing, as its first element, an indefinite-length array containing the remainder of the nesting, will incur storing a stack of size $\Omega(n)$. Thus, if a CBOR validator accepts such arbitrary nestings, then an attacker might exhaust its memory even with a *valid* byte sequence. For this reason, we choose not to support indefinite-length representations at all, and from now on, we only consider definite-length CBOR byte representations.

3.2 Formalizing Raw CBOR in PulseParse

We start by specifying and implementing a formally verified data validator, parser and serializer for raw CBOR bytes. Then, we use raw CBOR bytes specification to describe the CBOR data model, and we reuse the implementations for the deterministic subset of CBOR.

Specification and input validation We start with a specification for validating raw CBOR bytes. This can be done in constant stack space, using our validator for recursive formats, since CBOR meets the requirements of Theorem 2.1. CBOR item always starts with a *header* followed by a payload of other CBOR items, with nothing in between these CBOR items; and the header alone is enough to know how many CBOR items need to be validated in the payload (n for an array of n entries, $2n$ for a map of n entries, 1 for a tagged item, and 0 otherwise.) In fact, the header consists in the first byte, and the additional bytes needed to encode the number of array or map entries and the tag of a tagged item. For other items (integers, simple values, strings), the whole CBOR item is its own header, and the payload is empty. Our formalization starts by first defining a type of *raw CBOR data* shown (partially) below, representing maps as lists of pairs, and raw integers paired with a bound on their size in bytes.

```
type raw_u64 = { size:nat{ size ≤ 4 }; v:U64.t { fits_in size v }; }
type raw_data =
| Int64: (t:U8.t {t=0 ∨ t=1}) → (v:raw_u64) → raw_data
 . . .
| Map: (len:raw_u64) → (v:nlist (raw_data & raw_data) len.v) → raw_data
```

```
let raw_u64_prop (size:nat) (value:U64.t) =
 if size = 0 then value ≤ 23
 else value < pow2 (8 × pow2 (size − 1))
type raw_u64 =
{  size:  nat   { size ≤ 4 };
   value: U64.t { raw_u64_prop size value };  }
type raw_data =
| Simple: (v:U8.t { v≤23 ∨ v≥32 }) → raw_data
| Int64: (t:U8.t {t=0 ∨ t=1}) → (v:raw_u64) → raw_data
| String: (t: U8.t {t=2 ∨ t=3}) → (len:raw_u64) →
   (v: Seq.lseq U8.t len.value
    {t=3 ⟹ UTF8.correct v}) → raw_data
| Array: (len:raw_u64) →
   (v:nlist raw_data len.value) → raw_data
| Map: (len:raw_u64) →
   (v:nlist (raw_data & raw_data) len.value) → raw_data
| Tagged: (tag:raw_u64) → (v:raw_data) → raw_data

let parse_header = ...
let count_payload (x: raw_data) = match x with
| Array len _ → len.value
| Map len _ → 2 × len.value
| Tagged _ _ → 1
| _ → 0
let synth_cbor = ...
let parse_raw_cbor = parse_rec parse_header count_payload synth_cbor
```

Fig. 4. F⋆ inductive type for raw CBOR data

The we apply parse_rec to raw CBOR bytes, where count_payload reads the header to compute the number of items of each case; parse_header is a simple parser for the header bytes, and synth_payload constructs a raw_data from the list of parsed items.

```
let parse_raw : parser raw_data =
 parse_rec parse_header count_payload synth_cbor where
let count_payload = function | Map len _ → 2 × len.v | . . .
```

Thus, we prove that the parser specification for raw CBOR data is injective: raw CBOR bytes are a unique representation of raw CBOR data. This is true because the raw CBOR data type records all integer byte sizes and retains the order of all map entries. For any parse_rec, PulseParse by construction provides a corresponding low-level implementation combinator for validation, and since the CBOR header validator and the expected payload count function run in constant stack space, then so does the raw CBOR byte validator.

Parsing Concretely, we do not want to parse raw CBOR bytes into raw_data, since the latter is recursive and doing so would incur heap allocations. Instead, we provide an implementation-level parser iparse_raw which parses an input array of bytes into a low-level data structure of type iraw_data, which contains a partial parse of the input, with all the recursive occurrences represented simply by pointers into the input array. As such, we implement verified, incremental, mostly zero-copy parsing, in the sense that we do not copy variable-size data, but we only copy a constant amount of memory for one given call of the raw parser: such a call is not recursive and will only stack-allocate a constant amount of memory.

We provide *accessors* to inspect the contents of an iraw_data, e.g., for an array or a map, iparse_raw reads only its entry count, and we provide an accessor to iterate over the contents: calling the accessor will run iparse_raw once on the current array entry, or once on the current map entry key and once on the value. For an integer, iparse_raw reads it. For a string, iparse_raw reads its length, and provides a pointer to its payload. For a tagged item, iparse_rawr reads its tag, and we provide an accessor to access its payload: calling the accessor will run iparse_raw once on the payload.

To specify the correctness of iparse_raw, we define a relation (l:iraw_data) ↑ (h:raw_data) : slprop, relating a low-level partial parse l:iraw_data to a fully parsed high-level value h:raw_data. The triple below specifies iparse_raw:

```
{input ↦ b ∗ (|b| = n ∧ valid(b)) }
let res : iraw_data = iparse_raw (input, n)
{∃ (h:raw_data). (res ↑ h >∗ input ↦ b) ∗ parse_raw(b)==h }
```

The precondition says that, before running the parser, input points to some byte sequence b, and that b is of length n and starts with valid raw CBOR bytes. The postcondition shows that one gains access to a low-level result res corresponding to the high-level parse of b, and can give up access to res to regain ownership of input.

In full generality, our relation l ↑ r is equipped with fractional permissions [Boyland 2003], allowing shared readable access to parsed data. So, one can split res ↑ h, apply an accessor to res by using one fraction, leaving the other fraction available to apply other accessors if needed, and reconstitute the original full permission when one no longer needs the accessed data.

Serialization Whereas using accessors on iraw_data is enough to read them without paying much attention to the actual data structures, this assumption no longer holds for serialization. Indeed, we assume that an application will not try to serialize everything in the right order using fine-grained serialization combinators; instead, our definition of iraw_data, in the array and map cases accommodates a union of two cases, allowing to mix unparsed raw CBOR bytes (produced by iparse_raw) and recursive occurrences of iraw_data built by the application. Then, we build a recursive serializer for such raw CBOR data, where recursion is needed only for application-built items, and user-controlled unparsed bytes are copied as is. Thus, the recursion stack depth is entirely controlled by the application.

On the specification side, we define a recursive item serializer specification and we prove it correct with respect to the corresponding parser. Since the parser is injective, then the serializer is also injective. The implementation combinator takes an i:iraw_data, an output byte array and its length, serializing i into the output and returning the number of bytes written, or 0 if the output buffer is too small. We also implement a function computing the size of the raw byte representation, without serializing it.This takes as argument a piece of raw CBOR data, and an upper bound (to protect against arithmetic overflows), and returning the size of the byte representation, or 0 if it is larger than the bound. The implementations of the two functions have the exact same structure, apart from the actual output.

3.3 Specifying and Implementing the CBOR data model

We refine the raw CBOR model of the previous section first to CBOR (ensuring that maps have no duplicates) and then to Deterministically Encoded CBOR (ensuring that map keys are sorted, and that integers are represented minimally). For space reasons, we focus primarily on our main result that Deterministically Encoded CBOR is non-malleable and can be validated in constant stack space.

It is not enough to consider map key duplicates using mere equality on raw CBOR data. The major complication comes from the fact that maps can appear anywhere, including in keys; thus, to compare keys, we need to know how to compare maps within those keys, and to even compare those maps, we first need to know that those maps are themselves valid. In this process, we need to forbid a map from having two entry keys of equivalent representations, whether with integer representations of different sizes, or by the order of the map entry keys of the key itself.

To this end, on the specification side, we define two mutually recursive predicates: for a raw CBOR data to be *valid*, and for a pair of raw CBOR data to be *equivalent*. A piece of raw CBOR data x is valid if, and only if, all of its children data items (tagged payload, array items, map keys and values) are valid and, if x is a map, no two entries have equivalent map keys; and two pieces of raw CBOR data are equivalent if, and only if, they are equal, or both valid and of the same type, and, depending on their type, their integer values or simple values are equal (regardless of their byte sizes), or their array items or tag payloads are equivalent, or they are both key-value maps and their map entries seen as dictionaries associate equivalent keys to equivalent values.

To typecheck these predicates in F*, we have to prove that the recursion is well-founded. To this end, we first define the *size* of a raw CBOR data by structural recursion: a raw CBOR array has size 2 plus the sum of the sizes of its elements; a raw CBOR map has size 2 plus the sum of the sizes of its entry keys and the sizes of its entry values, a raw tagged CBOR data has size 2 plus the size of the payload; any other raw CBOR data has size 1. Then, we mutually define the validity and equivalence predicates by recursion on the sum of the sizes of their arguments.

In spite of the recursive nature of its specification, the validity of a piece of raw CBOR data x can be implemented in a way similar to the validator (or the jumper) for raw CBOR bytes, by maintaining a counter for the number of remaining children items to visit. This loop alone eliminates the need for a stack for the purpose of this visit. Moreover, checking for map key duplicates can be performed by two loops, one over the whole map, and one over the entries following the current entry for which we need to check that there are no other entries with an equivalent key. Thus, stack consumption only depends on the stack consumption of equivalence checking.

However, equivalence checking in general cannot be performed in a similar way, because of the order in map entries: checking the equivalence of two maps requires a stack at least proportional to the level of their map nesting.

No maps in map keys If a raw CBOR data has no maps in map keys, its validity can be checked in constant stack space, because, equivalence of map keys themselves containing no maps can be checked in constant stack space. This is enough for the COSE message layer, which mandates that map keys can only contain text strings and integers [Schaad 2022, § 1.5]. However, while this proves that validity for this subset of CBOR can be checked in constant stack space, this is not enough to define a formal data model for the whole CBOR.

Deterministically Encoded CBOR Fortunately, this limitation on map entry keys is not necessary, thanks to the "deterministic" encoding of CBOR relying on minimal integer byte sizes and map key ordering.

Given a total order on raw CBOR data, we first prove, by recursion on the sizes of their input CBOR data, that a piece of CBOR data where all of its integer byte sizes are minimal and all of its map keys are sorted with respect to the strict order, is valid; and two such pieces of CBOR data that are equivalent to each other are equal. However, this is not enough to prove that this representation covers all possible CBOR items. So, we prove, by induction on the size, that minimizing the integer byte sizes of valid raw CBOR data headers (integer value, tag value, or array or map entry count) x yields a valid raw CBOR data equivalent to x. Then, we prove that sorting the entries in a valid map where integers have minimal representation results in a valid, equivalent map. Thus, on the specification side, any valid raw CBOR data can be turned into such a representation, by recursively minimizing all its integer byte representations and sorting all its maps. Thus, we obtain the following:

Theorem 3.1. *Given a total strict order $<$ on raw CBOR data, the type* cbor *of raw CBOR data with minimal integer byte representations and maps sorted with respect to $<$ is a data model for CBOR, in the sense that there is a bijection between* cbor *and the following view type:*

$$
\begin{array}{lcl}
\mathit{view} & ::= & \mathit{Int}\ (x \in [-2^{64}; 2^{64}-1]) \\
& \mid & \mathit{Simple}\ (x \in [0,23] \cup [32,255]) \\
& \mid & \mathit{ByteString}\ (n \in [0, 2^{64}-1], x \in [0,255]^n) \\
& \mid & \mathit{TextString}\ (n \in [0, 2^{64}-1], x \in [0,255]^n \cap \mathit{UTF\text{-}8}) \\
& \mid & \mathit{Tagged}\ (\mathit{tag} \in [0, 2^{64}-1], v : \mathrm{cbor}) \\
& \mid & \mathit{Array}\ (n \in [0, 2^{64}-1], x \in \mathrm{cbor}^n) \\
& \mid & \mathit{Map}\ (n \in [0, 2^{64}-1], x :\ (\mathrm{cbor} \xrightarrow{n} \mathrm{cbor}))
\end{array}
$$

and there is a function size : cbor $\longrightarrow \mathbb{N}$, *such that a CBOR item has always strictly larger size than any CBOR item appearing in its view as its tagged payload, or an array or map entry.*

This view type is similar to but different than the mathematical data model of Fig. 4: the view type is not recursive, it is rather meant as a way to case analyze on a CBOR data item, where tag, array and map payloads are CBOR data items instead of views. But this time, maps are true mathematical finite maps with no key duplicates, and any integer byte sizes have disappeared.

The existence of the *size* function with the property on the view ensures that there are no *cyclic* CBOR items (e.g. an item that would appear itself in one of its tagged payloads, array or map entries.)

We instantiate this theorem with the lexicographic ordering on the byte representation of raw CBOR data with respect to the serializer specification defined in § 3.2. Then, since that serializer is injective, "Deterministically Encoded CBOR" is indeed a unique representation of CBOR items.

On the verified implementation side, we implement a function checking that raw CBOR bytes have minimal integer byte sizes and have their map entries sorted with respect to a strict order. The structure of this checker is similar to that of the jumper, where the stack consumption only comes from the function that compares two map keys. With the lexicographic byte ordering, stack consumption is constant.

However, on the serialization side, instead of implementing a function that would recursively sort map entries from valid raw CBOR bytes, we provide a verified, defensive API that allows constructing CBOR items using C or Rust data structures. As part of our verified API, we provide a function to create a CBOR map from an array of pairs of CBOR items representing the map entries. This function sorts the map entries in place without serializing them, thanks to the following theorem reflecting the lexicographic byte representation order at the level of the CBOR item view:

Theorem 3.2. *Let x_1 and x_2 be two CBOR items of respective types (as defined in Figure 2)t_1 and t_2. $x_1 < x_2$ with respect to their deterministic byte representation if, and only if, $t_1 < t_2$, or $t_1 = t_2$ and one of the following holds:*

(1) *they are both nonnegative integers, or simple values, and their values are ordered: $n_1 < n_2$*
(2) *they are both negative integers, or simple values, and their values are counter-ordered: $-1 - n_1 < -1 - n_2$*
(3) *they are both tagged items, and their tags $tag_1 < tag_2$, or $tag_1 = tag_2$ and their payloads $x'_1 < x'_2$*
(4) *they are both array items, and their number of entries $n_1 < n_2$, or $n_1 = n_2$ and their lists of entries are lexicographically ordered with respect to <*
(5) *they are both map items, and their number of entries $n_1 < n_2$, or $n_1 = n_2$ and their lists of entries, with the keys sorted wrt. <, are lexicographically ordered with respect to the lexicographic order on key-value pairs derived from <*

This theorem, leveraging big-endian encoding of integers of a given size, justifies the use of the lexicographic byte ordering over the length-first byte ordering defined in the previous version of the CBOR standard [Bormann and Hoffman 2013].

Our map creation function is defensive, in the sense that if it encounters duplicate keys during sorting, it gracefully fails.

Then, since map entries are sorted in their data structure representations, it is enough to reuse the raw CBOR data serializers that we defined in § 3.2, using minimal byte representations for integers, provided that user-controlled unparsed CBOR bytes use the deterministic encoding. Indeed, we deem this proviso necessary for security, because replacing bytes representing valid CBOR data with their deterministic encoding would need to be performed in depth-first fashion, thus requiring stack usage at least proportional to the depth of the CBOR item.

Calling the serializer returns the byte size of the binary representation, or 0 if the output buffer is too small, as specified as the following separation logic triple:

```
{ x ↑ v ∗ b ↦ s } let n = iserialize x b
{ ∃ s'. x ↑ v ∗ b ↦ s' ∗ ((n>0 ⟺ |serialize(v)| ≤ |s|) ∧
  (n>0 ⟹ (n=|serialize v| ∧ prefix n s'=serialize(v)))) }
```

We generate C and Rust serializers with the following signatures:

```
size_t iserialize(icbor x, uint8_t *output, size_t output_len);
fn iserialize <α>(x: icbor <α>, output: &α mut [u8]) → option_size_t
```

As such, one can use EverCBOR directly from C or Rust, as a high-assurance, full-featured CBOR library. Even among unverified implementations of CBOR, QCBOR, a “commercial-grade” implementation, has long not supported sorting of map keys until version 2.0, released in February 2025, and which is still alpha as of April 2025, thus illustrating the intricacies of implementing the deterministic encoding.

Limitations CBOR also allows representing floating-point numbers in IEEE 754 [IEEE 2019] half-precision, single-precision and double-precision formats. However, we do not support floating-point numbers, due to lack of F★ support, although formalizations of floating-point values and their representations exist for other theorem provers, such as Flocq for Coq [Boldo and Melquiond 2011]. Moreover, the statement of Theorem 3.2 for floating-point values would not be as simple as for integers, since the size prefix for floating-point values in the CBOR binary encoding indicates precision rather than magnitude. CBOR also provides for definition of further types (long integers, dates, etc.) as an interpretation of byte strings tagged with certain tags. Long integer representations potentially overlap with the standard representations of 64-bit integers, and a deterministic encoding allowing to conflate such representations would actually further *restrict* the space of valid byte representations. We leave such extended data models to future work.

4 EverCDDL: Verified Parsers and Serializers for CDDL

Although CBOR, like JSON, was initially being designed as a schema-less binary representation, most security-critical applications do not use CBOR as is, but rather want to parse and serialize CBOR items following a schema of their choice. To this end, in 2019, the IETF proposed CDDL (Concise Data Definition Language, [Birkholz et al. 2019]) as a schema language for CBOR. While CDDL is still a proposed standard, it has increasingly been used in other standards such as COSE [Schaad 2022], DPE [Trusted Computing Group 2023], and SCITT [Birkholz et al. 2025].

In this section, we introduce EverCDDL, a formal model of CDDL in F★, and a code generator that transforms a CDDL description to low-level types, parsers, and serializers for CBOR items valid with respect to such a description.

4.1 Syntax and Semantics

A simplified syntax for CDDL descriptions is shown below:

type	t	$::= \theta \mid [a] \mid \{g\} \mid t_1/t_2$
base	θ	$::= \bot \mid \ell \mid \texttt{any} \mid \texttt{int} \mid \texttt{uint} \mid \texttt{nint} \mid \texttt{tstr} \mid \texttt{bstr}$
label	ℓ	$::= n \in \left[-2^{64}, 2^{64}-1\right) \mid s : \text{UTF-8}$
array group	a	$::= t \mid a_1 \mathbin{//} a_2 \mid ?a \mid a_1, a_2 \mid *a$
map group	g	$::= t_k \Rightarrow t_v \mid \ell : t \mid g_1 \mathbin{//} g_2 \mid ?g \mid g_1, g_2 \mid *g$

We explain with an example: Two entities, a company and a non-profit, want to produce a record of their name, their status, and the names and salaries of its employees, encoding as a CBOR item which would have the following JSON shape:

```
[ "ACME Corp.", "company", { "J.D.": 1842, "M.S.": 1729, "CEO": "J.D." } ]
```

["The Main St. Assoc.", "nonprofit", { "John S.": 0 }]

Such CBOR items satisfy the following CDDL schema:

[tstr, ("company" / "nonprofit"), { ? ("CEO": tstr), * (tstr => uint) }]

matching an array of three CBOR items, the first being a text string for the entity name, the second being either "company" or "nonprofit" as a text string, and the third being a map containing an optional key-value entry with key equal to the text string "CEO" and a text string value, and zero or more key-value entries where keys are text strings for employee names, and values are nonnegative integers for their salaries.

Types In EverCDDL, we specify a CDDL type as a Boolean predicate on CBOR items: taking the cbor type defining the data model of Theorem 3.1, the semantics of a CDDL type is a Boolean function cbor → bool. For each CDDL type construct, we define its semantics as a predicate combinator. The standard dictates that the semantics of CDDL is with respect to CBOR without presumption of deterministic encoding—so, one cannot, assume, say, that map entries are ordered. Of course, CDDL can be and is used with Deterministically Encoded CBOR for security-critical applications.

Array groups An array group is one of: a type to describe a single CBOR element satisfying that type, an alternative choice of two array groups, an optional array group, a concatenation of two array groups, or a finite repetition of an array group (the Kleene star), which is interpreted in a greedy fashion, similarly to Parsing Expression Grammars (PEG) [Ford 2004a]. Thus, if a is an array group that consumes at least one CBOR item, then $*a, a$ will never match, since the first $*a$ will have consumed all sublists matching a, leaving none matching the second a. For a given array group a, the CDDL array type with array group a matches a CBOR item x if, and only if, x is a CBOR array and a consumes all of its entries.PEG semantics prescribe that the alternative is non-backtracking: $(a_1 \mathbin{//} a_2), a$ is not equivalent to $(a_1, a) \mathbin{//} (a_2, a)$ in most cases, unless a always succeeds. Consider a CBOR item list l, and assume that a_1 succeeds on l and returns remaining list l'. Then, if a fails, the whole array group fails, and the alternative a_2 will not be rechecked on l again. In EverCDDL, we specify an array group as a function that takes a list of CBOR items and returns a splitting pair of such a list, consisting of the list of consumed items and the list of remaining items; or None if the CBOR item does not match.

Map groups A map group is one of: an entry descriptor consisting of a type for the entry key and a type for the entry value; or an optional map group; or a finite repetition of a map group. An entry descriptor can be equipped with a *cut*, which is meant to be the last possible matching rule for keys matching the key type,in the sense that if there is an entry whose key matches the key type but the value does not match the value type, then the whole map fails to validate, regardless of alternatives. For instance, the map $(18 \mapsto 21)$ matches $?(18 \Rightarrow 42)$, with no entry consumed; but it does not match $?(18 : 42)$, because of the use of the cut ':' rather than '⇒'. Since that cut is nested within an option ?, its behavior is best described as an "exception" semantics. But since CDDL alternatives are not backtracking, there is no way to "catch" such an exception in a CDDL schema.Just like array groups, a map group can be seen as a function taking a map, potentially consuming some of its entries, and returning the map of unconsumed entries, with concatenation being function composition.

Deterministic map groups Unfortunately, not all map groups are admissible in CDDL, since some of them can be ambiguous because CBOR map entries are, in general, unordered. Consider the CBOR map $(18 \mapsto \text{"foo"}); (42 \mapsto \text{"bar"})$: the map group $(\texttt{uint => tstr})$ may match either of the two entries. Our semantics first defines the nondeterministic validity semantics of a map group as a function that takes a finite CBOR map and returns either a *set* of possible consumed-remaining map pairs, or ⊥ if a cut fails. Then, a map group is *deterministic* if, and only if, it returns ⊥ or a singleton set. We prove that, if t_k and t_v are CDDL types, then, even though $t_k \Rightarrow t_v$ may be nondeterministic, $*(t_k \Rightarrow t_v)$ is always deterministic, always succeeds, and consumes all map entries whose keys match t_k and values match t_v. We prove the following theorems.

Theorem 4.1. *If $t_1^k, t_1^v, t_2^k, t_2^v, \ldots$ are CDDL types, and $o_1, o_2, \cdots \in \{\Rightarrow, :\}$, then $*((t_1^k\ o_1\ t_1^v) \mathbin{//} (t_2^k\ o_2\ t_2^v) \mathbin{//} \ldots)$ has the same validity semantics as $*(t_1^k\ o_1\ t_1^v), *(t_2^k\ o_2\ t_2^v), \ldots.$*

Theorem 4.2. *The subset of CDDL map groups defined as follows yields only deterministic map groups:*

$$g ::= \ell \Rightarrow t \mid \ell : t \mid *(t_k \Rightarrow t_v) \mid g_1 \mathbin{//} g_2 \mid ?g \mid g_1, g_2$$

Type interpretation Every CDDL type t can be interpreted as type in F*, $[\![t]\!]$. For instance, $[\![\texttt{uint}]\!]$ is U64.t, the type of unsigned 64-bit integers; $[\![t_1/t_2]\!]$ is either$[\![t_1]\!][\![t_2]\!]$, the disjoint union. Similarly, we turn array or map group concatenation into a pair; the Kleene star for array groups as a list; and the Kleene star for map groups as the type Map.t key (list value), finite associations, accommodating duplicate keys with unspecified key ordering (subsequently, refined to forbid duplicates); and constant literals to the unit high-level type. For our illustrative example, the high-level type associated to an entity record is a tuple with a string for the entity name; either unit unit corresponding to the company or nonprofit alternative; option(unit & string) for the optional CEO field, and Map.t string (list U64.t) for the employee name-salary table.

Ambiguity The type interpretation exposes other challenges with ambiguity as well. For instance, CDDL does not require alternatives to be disjoint. Consider for instance the CDDL type uint/any. However, if we naively serialize the value Inr(Int(42)), which is the right-hand-side of the disjoint union type and parse it back, the parser could return Inl(42). As another example, consider the following CDDL map group $(18 \Rightarrow \texttt{uint}), *(\texttt{uint} \Rightarrow \texttt{any})$. If we try to serialize the high-level value $((() \mapsto [42]), (18 \mapsto [21]))$, the serializer should fail because the two CBOR maps obtained for each part of the concatenation will have non-disjoint domains, so it is impossible to concatenate those CDDL maps. To identify and rule out such ambiguities, we define an internal elaboration system for CDDL, which we describe next.

Elaboration Our elaboration of CDDL uses the extended syntax of deterministic map or map groups (shown below), with decorations on its domain, where $*((t_k \backslash t_{\text{rej}}) \Rightarrow t_v)$ is a table matching entries whose keys match t_k but not t_{rej} and values match t_v.

$$g ::= \ell \Rightarrow t \mid \ell : t \mid *((t_k \backslash t_{\text{rej}}) \Rightarrow t_v) \mid g_1 \mathbin{//} g_2 \mid ?g \mid g_1, g_2$$

$$\frac{}{(t;(t_k \Rightarrow t_v)) \rightsquigarrow (t/t_k;(t_k \Rightarrow t_v))}$$

$$\frac{}{(t;(\ell : t_v)) \rightsquigarrow (t/\ell;(\ell : t_v))}$$

$$\frac{}{(t;?(\ell : t_v)) \rightsquigarrow (t/\ell;?(\ell : t_v))}$$

$$\frac{(t/\ell;g_1) \rightsquigarrow (t_1;g_1') \qquad (t/\ell;g_2) \rightsquigarrow (t_2;g_2')}{(t;((\ell : t_v),g_1) \mathbin{/\!/} g_2) \rightsquigarrow (t_1 \cap t_2;((\ell : t_v),g_1') \mathbin{/\!/} g_2')}$$

$$\frac{(t;g_1) \rightsquigarrow (t_1;g_1') \qquad (t;g_2) \rightsquigarrow (t_2;g_2')}{(t;g_1 \mathbin{/\!/} g_2) \rightsquigarrow (t_1 \cap t_2;g_1' \mathbin{/\!/} g_2')}$$

$$\frac{(t;g_1) \rightsquigarrow (t_1;g_1') \qquad (t_1;g_2) \rightsquigarrow (t_2;g_2')}{(t;g_1,g_2) \rightsquigarrow (t_2;g_1',g_2')}$$

$$\frac{}{(t;*(t_k \Rightarrow t_v)) \rightsquigarrow (t;*((t_k \backslash t) \Rightarrow t_v))}$$

Fig. 5. Annotating map group tables with excluded sets of keys. For two types t_1, t_2, we compute an underapproximation $t_1 \cap t_2$ of their intersection.

Elaboration $elab(t)$, is a partial function, proceeding in several steps. First, we use Theorem 4.1 to rewrite map groups into a canonical form, and then check that map groups are all of the deterministic form of Theorem 4.2. If not, we reject the specification.

Next, for each deterministic map group g, we annotate its tables with key type specifications that should be rejected. To this end, we define the function $(t;g) \rightsquigarrow (t';g')$, defined in Figure 5, saying that a map group g applied to any map that has no keys matching t behaves the same as g', and if successful, the remaining map entries have no keys matching t'. The rewrite rules are specified in priority order, so the fourth rule takes precedence over the overlapping fifth rule. For a given map descriptor $\{g\}$, we rewrite $(\bot, g) \rightsquigarrow (t', g')$, and use g' as its elaborated form. Finally, we check the following properties, rejecting g' if any of them fail: (1) all alternatives must be disjoint; (2) for any array groups a_1 and a_2, if $*a_1, *a_2, a_3$ appears, then a_1 and a_2 must be disjoint and a_1 and a_3 must be disjoint; and if $*a_1, a_2$ appears, then a_1 and a_2 must be disjoint. (This is to avoid things like $*a, a$, which we know will never match); and (3) for any map groups g_1 and g_2, if g_1, g_2 appears, then the footprints of g_1 (the types of all keys appearing in g_1, minus the excluded keys t_{rej} in $*((t_k \backslash t_{\text{rej}}) \Rightarrow t_v)$) and g_2 must be disjoint.

Theorem 4.3. *Given a CDDL type t, if $elab(t) = t'$ is defined, then t and t' have equivalent validating semantics: a CBOR item is valid for t if and only if it is valid for t'.*

We also prove that elaborated types are unambiguous, though first we need to introduce the semantics of CDDL parsers.

The elaboration described above is a simplification: indeed, we have extended the implementation of EverCDDL to annotate tables with key-value footprints (instead of just keys), represented as Boolean formulae where atoms are pairs of key-value types. This allows us to support extensibility patterns such as $?(18 \Rightarrow$ `uint`$), *($`uint` $\Rightarrow$ `any`$)$, where the table $*($`uint` $\Rightarrow$ `any`$)$ can accept an entry with key 18, provided its value is not an unsigned integer.

Parsing Semantics A main design goals of CDDL is to "enable extraction of specific elements from CBOR data for further processing" [Birkholz et al. 2019, § 1], which basically means parsing. The parsing specification of a CDDL type t is a function taking a CBOR item, item list or map valid with respect to t, and returning a value of type $[\![t]\!]$. For instance, for `uint`, the parser specification extracts the integer value of a CBOR item an returns it as a U64.t. For t_1/t_2, the corresponding parser is $p(x) = \mathsf{Inl}(p_1(x))$ if x satisfies t_1, and $\mathsf{Inr}(p_2(x))$ otherwise, where p_i is the parser for t_i.

This brings us to our main theorem about the semantics of CDDL:

Theorem 4.4. *Given a CDDL type t, if $elab(t)$ is defined, and p is the parser specification associated with t, then p is injective; we can define a* serializability *function $\sigma : u \to bool$, such that for any CBOR data x valid with respect to t, $\sigma(p(x))$ holds; and we can define a serializer specification $s : (x : u\{\sigma(x)\}) \to cbor$ such that for any serializable high-level value x, $p(s(x)) = x$.*

The serializability function σ refines the type interpretation $[\![t]\!]$ to enforce constraints such as the absence of duplicate keys in maps.

Extensions and limitations We have presented a simplified version of what EverCDDL supports. In particular, our implementation also supports integer ranges, byte lengths, and UTF-8 strings.

We only support non-recursive CDDL descriptions; while we investigated the formal semantics of recursive CDDL descriptions, we ultimately deem them a security issue because they would give rise to stack consumption proportional to the size of the input. Standards such as COSE use recursion only up to a depth of 2 or 3, which is easily supported by unrolling.

4.2 Code Generation: Implementing Formatters for CDDL

Once EverCDDL elaborates and proves the unambiguity of a CDDL definition, it generates implementation code in Pulse for types, validators, parsers, and serializers.

Validators A validator for a CDDL type t takes as argument a CBOR item (obtained either from calling the EverCBOR parser, or by constructing a CBOR item using the EverCBOR API) and returns a Boolean value, `true` if and only if the CBOR item is valid with respect to t. For CDDL array groups, the validator takes as argument a pointer to a CBOR array iterator (the pointer is stack-allocated by the caller) and returns `true` if and only if the array group succeeds, with the validator advancing the iterator to consume the relevant array items. For CDDL map groups, the validator takes as argument a CBOR item representing a map, and a caller-allocated pointer to the number of map entries that have not been consumed yet. Since the validators rely on the fact that EverCDDL only concatenates map groups with disjoint key domains, it is enough to count the number of map entries left, and there is no need to precisely track which entries have been consumed. Thus, validating a map does not require any heap allocation, though incrementally validating the entries of a map may require repeatedly scanning a prefix of already validated keys.

Parsers The parser implementation for t takes a CBOR item assumed to be valid with respect to t, and returns a low-level representation $l : [\![\hat{t}]\!]$ of the high-level value $h : [\![t]\!]$ returned by the parser specification, similar to the definition of iparse_raw in §3. The difference is that EverCDDL also generates the $l \uparrow v$ separation logic predicate relating low-level and high-level values. At the top-level, we combine the EverCBOR validator and parser with the EverCDDL validator and parser, producing a function that takes as input a byte array and its length, and returns a low-level representation of the result of the CDDL parser specification and the remainder of the byte arraypast the byte representation of the corresponding CBOR item, or `None` if the input bytes are not a valid representation of a CBOR object valid with respect to t.

For a given array group g, the parser implementation takes as argument a caller-allocated pointer to a CBOR array iterator assumed to be valid with respect to g, and returns a low-level representation of the high-level value returned by the parser specification. If g is a Kleene star $g = *g'$, then, similarly to EverCBOR, we do not parse the full contents of the array. Rather, we split the array iterator into two adjacent slices, the left-hand-side one covering all array items consumed by g; then we return that iterator slice along with a function pointer to the array parser for g', leaving to the application the responsibility of advancing that iterator to parse the array elements. Map groups are similar, where for a table, we do not parse the full contents of the map. Rather, we return a record value containing the CBOR map and function pointers for the validator for the CDDL key type, the key exclusion domain, and the value type, as well as parsers for the key and value types. The validator function pointers are necessary since matching map entries are not necessarily contiguous, contrary to arrays. We then provide a generic iterator combinator to advance the map accordingly.

Serialization Contrary to parsing, we generate serializers that directly produce the deterministic byte encoding of the CBOR item that is the result of the serializer specification, rather than producing a CBOR data by allocating intermediate `iraw_data` objects for use with the EverCBOR API. A serializer for t takes as argument a low-level $l : [\![\hat{t}]\!]$, an output byte array and its length, and returns the number of bytes written, or 0 if the output array is too small or if the high-level value is not *serializable* (e.g., it violates the serializability condition σ from Theorem 4.4 with integer or simple value out of bounds, invalid UTF-8 text bytes, etc.)

For the array descriptor and the map descriptor, the serializer first calls the array group or map group serializer, then encodes the header with number of entries written, then swaps the entries and the header. This is necessary for the deterministic encoding if we want to traverse the input data at most once. An alternative could be to traverse the input data twice, once to compute the number of entries to write, and another one to serialize the entries. If we were not using the deterministic CBOR encoding, we could always use 9 bytes to store the number of entries (1 byte for the CBOR type, plus 8 bytes for the integer encoding, see Fig. 3)

For the Kleene star in array groups, we generate a serializer that takes as argument either an array of low-level representations of high-level values to serialize, or an array iterator that was the result of a parser. Thus, the serializer can serialize the contents of an array returned by another parser, provided the relations between the low-level array item representation and the high-level value match between the parser and the serializer. To this end, we strive to make the relation depend as little as possible on the parser specification.

For map groups, we generate a serializer that takes an output buffer already containing some map entries sorted with respect to the lexicographic byte order, and inserts serialized map entries into it, using sorted insert: for each entry to insert, the serializer first writes it next to the existing output map, then it scans the output map, comparing keys to determine where to insert the new entry, then it swaps the new entry with the tail of the output map that follows the insertion point. In doing so, it can detect that an entry with the same key already exists in the output map. In that case, the serializer gracefully fails. This is interesting especially for tables: this check on serialized output maps is a sound way to check that the *input* map has no duplicates. From the verification point of view, the high-level datatype is neither a map (because the serializer does not need to assume that the input map has no duplicate keys) nor a list of entries (because the serializer does not need to know about the order of entries), but a map between keys and lists of values: this way, keys are not ordered, but, for a given key, the length of the list of values equals the number of occurrences of the key in the "map".

5 Performance Benchmarks & Verified Applications

In this section, we report on experiments using our verified tools, with both quantitative and qualitative results. It's worth noting that our verified code worked correctly the first time on all experiments. On an Intel Xeon E5-2680 v4 with 56 cores (1.2 GHz), using 24 cores, PulseParse (650 lines for `parse_rec` and its proofs + 700 lines for its implementation + 6400 lines for all the Pulse combinators) verifies in 6 minutes, EverCBOR (6k lines of spec + 26k lines of implementation and proofs) verifies in 10 minutes and extracts and compiles to both C and Rust in 1 minute. Finally, EverCDDL (6k lines of spec + 23k lines of implementation and proofs) verifies in 0.5 hour.

Although we generate both C and Rust code, we focus on evaluating the performance of the generated C code, unless explicitly stated otherwise.

5.1 Synthetic Benchmarks

We evaluate EverCBOR and EverCDDL on several synthetic benchmarks, and show that its performance is comparable with that of existing (unverified) libraries, namely QCBOR and TinyCBOR, even though we have not had the time to implement any optimizations after these initial benchmarking results.

Our first benchmark considers a record type with 8 fields of type `uint`, with results in the first line of Table 1. From a CDDL description (elided), EverCDDL generates a struct type and parsers and serializers for it. The QCBOR and TinyCBOR libraries do not provide CDDL functionality, so we write C functions translating between the CBOR representation and a flat C structure. The performance of our validator and parser is between QCBOR and TinyCBOR. We believe we can close the gap to QCBOR since, by default, EverCDDL returns parsed records as structures on the stack, rather than using out parameters to fill an existing object—it should be straightforward to add support for this and close the gap to QCBOR. For serialization, our code is slower than QCBOR and TinyCBOR because we serialize in the deterministic encoding, perhaps shuffling elements.

Our second benchmark involves large maps. The relevant CDDL description is simply `map = *(uint => uint)`. The benchmark consists of a map with $N = 8000$ entries with random keys and values, which is then looked up $K = 1000$ times with random keys, which may or may not be present. We begin from a serialized map in the deterministic encoding. For EverCDDL, we first validate this bitstring, which checks that the keys are in order, obtaining an iterator. To look up a value, we construct a CBOR object from our desired key, and call an EverCBOR function to look it up in the map. The QCBOR API offers a function for map lookup, so we use it. For TinyCBOR, we iterate through the map comparing keys. Here, EverCBOR is faster than the other two libraries, for two main reasons. One, given that we validated the map, we know the keys are in order and can therefore stop early safely (we also stop early with TinyCBOR). Second, importantly, since we know the object is deterministically encoded, we can compare the serialized representation of keys directly, byte-for-byte, instead of having to parse back the keys in the map.

Our third benchmark involves nested arrays, generated by the CDDL description `arr = [*subarr]; subarr = [*uint]`. By running EverCDDL on this description, we generate a C type for an `arr`, alongside a parser and serializer for it. We measure the time it takes to serialize and parse an array of $N = 10^4$ where every subarray also contains N elements all set to zero, for a total of 10^8 elements. The CBOR object involved is roughly 100MB. For EverCDDL, we generate a structure in memory and call the serializer. The QCBOR library, instead, provides a streaming API where elements are output or parsed one at a time. We include the setup time in the measurements for EverCDDL, for a conservative comparison. EverCDDL is less performant than the other libraries, there are a few non-fundamental reasons for this. For example, when writing each integer into the buffer, there is a size check performed. This check involves constructing the CBOR object (of a single integer) to be written, computing its size, and checking that the remaining space is at least that. This computation is rather wasteful and hard to optimize by the C compiler. Specializing it manually, replacing the size of the CBOR integer by the constant 1, provides a 20% performance improvement. We are confident we can adjust our verified implementations to generate specialized sizes to attain this speedup.

For parsing, there is a design difference between the APIs provided by EverCDDL and the other two libraries. EverCDDL requires the buffer to be validated before any data can be read from it, which incurs one full pass of the 100MB buffer. Once validated, the client code can use the iterators to walk the object, without incurring copies, and extract the integers in it. The other libraries provide streaming APIs that can simply walk the buffer and read the integers on demand, avoiding the need for the initial pass, but allowing to partially read a corrupted object. For security-sensitive applications, we argue that a validation pass should be performed in all cases, hence our benchmark for QCBOR and TinyCBOR also include one such pass. For validation, all three libraries perform similarly. However, our parsing is slower, because the EverCDDL iterator for the outer array is not related at all to that of the inner array. Once the inner iterator reaches the end, and we want to advance to next subarray, the outer iterator has to walk the buffer again to find the new offset. Our current iterator API does not expose this fact, mainly because it treats unparsed and application-built data uniformly.

	EverCDDL		QCBOR		TinyCBOR	
	V/P	S	V/P	S	V/P	S
Rec (μs)	3.33	.57	1.91	.23	3.78	.29
Map (μs)	138		282		306	
Arr (s)	2.67/4.92	2.06	2.92/2.91	0.75	2.68/2.68	1.23

Table 1. Synthetic benchmarks for EverCDDL, QCBOR and TinyCBOR. Values are time (for Rec, for **V**alidation plus **P**arsing, or **S**erialization), lookup time (for Map), or time (for Arr).We distinguish validation from parsing in Arr, since iteration is involved.

	C API & OpenSSL	Pulse API & HACL⋆
COSE_sign	39.0 μs/iter	53.3 μs/iter
COSE_verify	99.6 μs/iter	58.2 μs/iter
Ed25519_sign	36.8 μs/iter	51.9 μs/iter
Ed25519_verify	96.7 μs/iter	57.3 μs/iter
parse(Sign1)	2.4 μs/iter	
ser(Sign1)	1.0 μs/iter	
ser(Sig_structure)	1.0 μs/iter	

Table 2. Benchmarking results of our EverCDDL-based COSE signature implementation. We sign and verify a message with an 896 byte long payload using Ed25519. The benchmarks were compiled with clang 19.1.7 (-O3) and run on an Intel Xeon W-2255 CPU.

All in all, while there are some improvements to be made, our benchmarks show performance close to a state-of-the-art unverified library, although our code parses to and from application-level types with a verified, defensive implementation.

5.2 Verified Applications: COSE & DPE

COSE is an Internet standard for signing and encryption of CBOR objects, initially for securing the transport of IoT messages, though it is also used today in non-IoT settings. For signing, COSE defines a signature envelope message format containing a signature structure. The signature structure is encoded using Deterministically Encoded CBOR to make sure its byte representation is unique; then, it is authenticated using cryptographic hashing algorithms.

In the COSE standard, the message formats are described normatively in prose, but they are accompanied with a non-normative CDDL description. We found that the latter does not reflect the normative prose on two aspects, namely the constraint that keys 5 and 6 must not appear together in Generic_Headers, and an erroneously backtracking (non-PEG) interpretation of ? in Sig_structure. So, we fixed the CDDL description accordingly.

With EverCDDL, we support signature and verification formats with a single (COSE_Sign1) or multiple signers (COSE_Sign), as well as some cryptographic key object formats.

A notable limitation of our implementation of COSE is that the parser only supports deterministic CBOR. Hence our implementation will reject COSE messages that are not deterministically encoded. This is not a problem when serializing messages since it is always allowed to write CBOR deterministically.

Evaluation The F★ file generated by EverCDDL for the COSE specification takes 5 minutes to verify on a single core; extracting to C using Karamel takes another 23 seconds. To evaluate interoperability and benchmark the performance of the EverCDDL-generated code, we implement a small signature generation and verification tool (limited to a single signer, Ed25519 algorithm, empty AAD, fixed headers) in two versions: both an unverified one using the EverCDDL-generated C API and OpenSSL, as well as a verified one using the Pulse API and using the HACL★ library for cryptographic operations. The benchmarking results in Table 2 show that the cryptographic primitives take up the majority of the runtime, in both the verified and unverified versions.[5]

To give a flavor of the automatically generated C API, let us look at the CDDL schema for `COSE_Key_OKP`. This type specializes `COSE_Key` in the COSE RFC to OKP keys; specializing the type makes EverCDDL parse the fields for the public key (-2) and private key (-4), and we do not need to go through the map manually.

```
COSE_Key_OKP = { 1:1, −1:int/tstr, ?−2:bstr, ?−4:bstr, *label=>values }
```

On the C side, we get a structure and two functions, for serialization and parsing.[6]The structure has four fields: three for explicitly specified data fields (-1, -2, and -4) and one for the map at the end. The entry `1 : 1` does not correspond to a field in the C structure, since EverCDDL knows it just has the value 1. Types like `option___bstr` are created by Karamel using monomorphization.

```
typedef struct {
  label intkeyneg1; option__bstr intkeyneg2; option__bstr intkeyneg4;
  either__slice__map_iterator_t _x0; } COSE_Key_OKP;
size_t serialize_COSE_Key_OKP(COSE_Key_OKP c, slice__uint8 out);
option__COSE_Key_OKP__slice_uint8
  validate_and_parse_COSE_Key_OKP(slice__uint8 s);
```

The Pulse API generated by EverCDDL is expressive enough to state a precise functional correctness specification. For signature verification, we define a predicate relating a valid signature message with its payload, where vmsg is the specification-level struct carrying the signed bytes, while tbs is the bytes to be signed:

```
let good_sig pubkey msg payload = ∃ vmsg tbs.
  parses_from bundle_COSE_Sign1_Tagged.b_spec vmsg msg ∧
  vmsg.payload == Inl payload ∧ length vmsg.sig == 64 ∧
  to_be_signed_spec vmsg.protected payload tbs ∧
  spec_ed25519_verify pubkey tbs vmsg.sig
```

The verify function then takes fractional (shared) permissions to the public key and (serialized) message, and returns an optional slice for the payload. The postcondition ensures that any payload returned by verify is signed by the given public key.

```
{ pubkey ↦(r) vk ∗ msg ↦(p) vm } let payload = verify pubkey msg
{ pubkey ↦(r) vk ∗ (match payload with | None → msg ↦(p) vm
  | Some r → ∃ vp q. (r ↦(q) vp) −∗ (msg ↦(p) vm) ∗ good_sig vk vm vq) }
```

[5]We were surprised that OpenSSL signature verification is three times slower than signing and also slower than the fully verified HACL★ implementation. The `t_cose` library however exhibits the same phenomenon (showing nearly identical performance), which is perhaps a sign of the complexity of using the OpenSSL API.

[6]We shorten namespaces in the generated C code.

Similarly, signature generates guarantees that the output buffer is a well-formed `COSE_Sign1_Tagged` object whose signature field is a valid signature of the appropriate `Sig_structure`.

DPE We also specify the CDDL API for DPE, a secure boot protocol, with functional correctness proofs, covering six different message types for the four main functions on the DPE interface. Appendix C provides some more information, though the main takeaway is similar to what we report for COSE: EverCDDL specifications are precise enough to express full functional correctness of application code manipulating messages in a given CDDL schema.

6 Related Work & Conclusions

Bratus et al. [2017] provide a useful perspective on the important of parsing for software security, including guidelines for how to securely handle attacker-controlled input.

The most closely related line of work to ours is EverParse [Ramananandro et al. 2019], which we have discussed throughout the paper, since we reuse some of their purely functional specification combinators. Many others have looked at purely functional verified parsers and serializers. Blaudeau and Shankar [2020] build a verified packrat parser for parsing expression grammars (PEGs) [Ford 2002, 2004b] in the PVS proof assistant [Shankar 1996], while [Mundkur et al. 2020] supports PEGs with constraints. Lasser et al. [2019] build a verified implementation of an LL(1) parser generator and Lasser et al. [2021] verified an implementation of the ALL(*) parsing algorithm, both in the Coq proof assistant. Ni et al. [2023] use EverParse's specification combinators to formalize ASN.1 DER [ITU-T Study Group 17 2021], a widely used data formatting standard with goals similar to CBOR and CDDL, proving that ASN.1 DER is non-malleable. They use an ad hoc approach to formalizing the recursion present in ASN.1, rather than our general purpose parse_rec combinator with constant-stack-space validation. Ni et al. extract their specifications to OCaml code, rather than going to fully low-level code in C, as we do. Similarly, Debnath et al. [2024] also focus on ASN.1 and formalize it in Agda, producing functional Haskell code for X.509 certificate chain validation. Delaware et al. [2019] implement a combinator library for verified parsers and serializers for binary formats in Coq, but they focus on producing purely functional programs in OCaml, rather than zero-copy, low-level code. They also do not prove non-malleability of formats.

Others have also looked at tools for low-level parsing and serializing. Bangert and Zeldovich's (2015) Nail is a DSL for writing low-level applications while processing a given data format. It produces C code, but does not aim at verification. Diatchki et al.'s (2024) Daedalus is a DSL with parser combinators targeting both Haskell and C++, aiming to produce memory safe C++, but without formal proof. Daedalus has been used at scale, including to generate parsers for the PDF document standard. Daedalus does not support serialization. Recent unpublished work describes a tool called Vest (https://github.com/secure-foundations/vest) a parser and serializer generator embedded in Verus [Lattuada et al. 2024], a dialect of Rust aimed at verification. Vest's use of linear types in Rust is similar in spirit to our use of separation logic. However, Vest does not support recursive formats which are required to formalize languages like CBOR. PulseParse is not tied to Rust, and Pulse can in general be used to produce verified C code or verified, safe Rust code, though support for the latter is not fully complete.

Conclusions In summary, we have presented a new approach to secure, low-level formatting with foundations in separation logic. We have used this foundation to develop a comprehensive, mechanized formalization of CBOR and CDDL, two data formatting standards of significant stature in security-related protocols, and applied our tools, including formally verified libraries and code generators, to a variety of other standards grounded in CBOR. We hope our open-source tools will help others build systems that process these binary formats correctly and securely.

References

Julian Bangert and Nickolai Zeldovich. 2015. Nail: A Practical Tool for Parsing and Generating Data Formats. *login Usenix Mag.* 40, 1 (2015). https://www.usenix.org/publications/login/feb15/bangert

H. Birkholz, A. Delignat-Lavaud, C. Fournet, Y. Deshpande, and S. Lasker. 2025. An Architecture for Trustworthy and Transparent Digital Supply Chains (SCITT). draft-ietf-scitt-architecture-11. https://www.ietf.org/archive/id/draft-ietf-scitt-architecture-11.txt

Henk Birkholz, Christoph Vigano, and Carsten Bormann. 2019. Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures. IETF RFC 8610. doi:10.17487/RFC8610

Clement Blaudeau and Natarajan Shankar. 2020. A Verified Packrat Parser Interpreter for Parsing Expression Grammars. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 3–17. doi:10.1145/3372885.3373836

Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *Proceedings of the 2011 IEEE 20th Symposium on Computer Arithmetic (ARITH '11)*. IEEE Computer Society, USA, 243–252. doi:10.1109/ARITH.2011.40

Carsten Bormann and Paul E. Hoffman. 2013. Concise Binary Object Representation (CBOR). IETF RFC 7049. doi:10.17487/RFC7049

Carsten Bormann and Paul E. Hoffman. 2020. Concise Binary Object Representation (CBOR). IETF RFC 8949. doi:10.17487/RFC8949

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.

Sergey Bratus, Lars Hermerschmidt, Sven M. Hallberg, Michael E. Locasto, Falcon Momot, Meredith L. Patterson, and Anna Shubina. 2017. Curing the Vulnerable Parser: Design Patterns for Secure Input Handling. *login Usenix Mag.* 42, 1 (2017). https://www.usenix.org/publications/login/spring2017/bratus

Andre Büttner and Nils Gruschka. 2023. Protecting FIDO Extensions Against Man-in-the-Middle Attacks. In *Emerging Technologies for Authorization and Authentication*, Andrea Saracino and Paolo Mori (Eds.). Springer Nature Switzerland, Cham, 70–87.

Joyanta Debnath, Christa Jenkins, Yuteng Sun, Sze Yiu Chau, and Omar Chowdhury. 2024. ARMOR: A Formally Verified Implementation of X.509 Certificate Chain Validation. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1462–1480. doi:10.1109/SP54263.2024.00220

Christian Decker and Roger Wattenhofer. 2014. Bitcoin transaction malleability and MtGox. In *European Symposium on Research in Computer Security*. Springer, 313–326.

Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.* 3, ICFP (2019), 82:1–82:29. doi:10.1145/3341686

Iavor S. Diatchki, Mike Dodds, Harrison Goldstein, Bill Harris, David A. Holland, Benoit Razet, Cole Schlesinger, and Simon Winwood. 2024. Daedalus: Safer Document Parsing. 8, PLDI, Article 180 (June 2024), 25 pages. doi:10.1145/3656410

Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 2025. PulseCore: An Impredicative Concurrent Separation Logic for Dependently-Typed Programs. In *46th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. (accepted for publication, to appear).

European Union eHealth Network. 2021. European Union Digital COVID Certificate (EUDCC) Electronic Health Certificates Specification. https://github.com/ehn-dcc-development/eu-dcc-hcert-spec.

Hal Finney. 2006. Bleichenbacher's RSA signature forgery based on implementation error. *https://mailarchive.ietf.org/arch/msg/openpgp/5rnE9ZRN1AokBVj3VqblGlP63QE/* (2006).

Bryan Ford. 2002. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 36–47. doi:10.1145/581478.581483

Bryan Ford. 2004a. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. Association for Computing Machinery, New York, NY, USA, 111–122. doi:10.1145/964001.964011

Bryan Ford. 2004b. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 111–122. doi:10.1145/964001.964011

Aymeric Fromherz and Jonathan Protzenko. 2024. Compiling C to Safe Rust, Formalized. arXiv:2412.15042 [cs.PL] https://arxiv.org/abs/2412.15042

Graham Hutton. 1989. Parsing Using Combinators. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag, Berlin, Heidelberg, 353–370.

IEEE. 2019. IEEE 754-2019 - IEEE Standard for Floating-Point Arithmetic. 84 pages. doi:10.1109/IEEESTD.2019.8766229

Intel. 2021. TinyCBOR. https://github.com/intel/tinycbor.

ITU-T Study Group 17. 2021. X.680 : Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation. ITU Recommendation X.680. https://www.itu.int/rec/T-REC-X.680/

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2019. A Verified LL(1) Parser Generator. In *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA (LIPIcs, Vol. 141)*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24:1–24:18. doi:10.4230/LIPIcs.ITP.2019.24

Sam Lasser, Chris Casinghino, Kathleen Fisher, and Cody Roux. 2021. CoStar: a verified ALL(*) parser. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 420–434. doi:10.1145/3453483.3454053

Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. 2024. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles* (Austin, TX, USA) *(SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 438–454. doi:10.1145/3694715.3695952

Laurence Lundblade. 2020–2023. QCBOR. https://github.com/laurencelundblade/QCBOR.

MITRE. 2016. CVE-2016-1494. https://www.cve.org/CVERecord?id=CVE-2016-1494.

Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. 2020. Research Report: The Parsley Data Format Definition Language. In *2020 IEEE Security and Privacy Workshops (SPW)*. 300–307. doi:10.1109/SPW50608.2020.00064

Haobin Ni, Antoine Delignat-Lavaud, Cédric Fournet, Tahina Ramananandro, and Nikhil Swamy. 2023. ASN1*: Provably Correct, Non-malleable Parsing for ASN.1 DER. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 275–289. doi:10.1145/3573105.3575684

Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified low-level programming embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (Aug. 2017), 29 pages. doi:10.1145/3110261

Tahina Ramananandro, Antoine Delignat-Lavaud, Cedric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1465–1482. https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 55–74.

Jim Schaad. 2022. CBOR Object Signing and Encryption (COSE): Structures and Process. IETF RFC 9052. doi:10.17487/RFC9052

Natarajan Shankar. 1996. PVS: Combining Specification, Proof Checking, and Model Checking. In *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1166)*, Mandayam K. Srivas and Albert John Camilleri (Eds.). Springer, 257–264. doi:10.1007/BFb0031813

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 256–270. https://www.fstar-lang.org/papers/mumon/

Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, and Arti Gupta. 2022. Hardening attack surfaces with formally proven binary format parsers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 31–45. doi:10.1145/3519939.3523708

The FIDO Alliance. 2025. Client to Authenticator Protocol (CTAP), version 2.2. https://fidoalliance.org/specs/fido-v2.2-ps-20250228/fido-client-to-authenticator-protocol-v2.2-ps-20250228.html.

Trusted Computing Group. 2023. DICE Protection Environment, Version 1.0, Revision 0.6. https://trustedcomputinggroup.org/wp-content/uploads/TCG-DICE-Protection-Environment-Specification_14february2023-1.pdf.

Word Wide Web Consortium. 2019. WebAuthn: An API for accessing Public Key Credentials. https://w3c.github.io/webauthn/.

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *ACM Conference on Computer and Communications Security*. ACM, 1789–1806. http://eprint.iacr.org/2017/536

A A Recursive Format: Variable Arity Trees

Consider for instance a small integer arithmetic language with numeric values, binary subtraction, and variable-arity addition. We specify this language as a high-level F* inductive datatype:

```
type expr = | Value of U64.t | Minus of (expr * expr)
| Plus: (n: U8.t {n<254}) → (l: nlist n expr) → expr
```

We bound the number of addition operands to 253 because we want to represent the node in the first byte: 255 for a value, followed by 8 bytes for the integer value; 254 for a subtraction, followed by 2 recursive payloads; otherwise, the object is an addition and the value of the first byte gives the number of recursive operand payloads.

To this end, we specify a header parser for elements using parser specification combinators:

```
let header = dtuple2 U8.t (λ h → if h = 255 then U64.t else unit)
let parse_header = parse_u8 `parse_dtuple2` (λ fb →
  if h = 255 then parse_u64 else parse_empty)
```

Note that the header contains the non-recursive integer value for the value case, but does not contain any recursive payload for the subtraction and addition cases.

Then, we define a F* function taking a header and determining the number of recursive payloads needed:

```
let count_payloads (h: header) = let (| fb, ob |) = h in
  if fb = 255 then 0 else if fb = 254 then 2 else fb
```

Then, we define a F* function to turn a header and a list of recursive expression payloads into an expression:

```
let synth (h: header) (pl: nlist (count h) expr) : expr = match h, pl with
| (| 255, v |), _      → Value v
| (| 254, _ |), [a; b] → Minus (a, b)
| (| n  , _ |), pl     → Plus n pl
```

Then, we can call the recursive parser combinator to obtain the parser specification for our expression language; thus enjoying validation in constant stack space.

Then, using the zero-copy reader combinators we defined in PulseParse, we implement a shallow parser performing case analysis on an expression implementation into the following implementation datatype, leaving recursive payloads unparsed:

```
type parsed_to = | PValue of U64.t | PMinus of byte_array * byte_array
  | PPlus: (n: U8.t) → (pl: byte_array) → parsed_to
```

This datatype extracts to C as a tagged union.

By contrast, since we assume applications to have full control of their memory consumption, we allow them to build arbitrarily nested expressions, potentially containing some unparsed data for some operands; thus, we provide the following implementation datatype from which to serialize:

```
type serialize_from = | SBase of parsed_to
| SMinus of ref serialize_from * ref serialize_from
| SPlus: (len: U8.t) → (pl: narray len serialize_from)
```

```
size_t validate(uint8_t *input, size_t len) {
  size_t expected = 1;
  size_t pos = 0;
  while (expected > 0) {
    expected = expected − 1;
    size_t header_size = validate_header(input+pos, len−pos);
    if (header_size == 0) return 0;
    pos = pos + header_size;
    if (expected > len − pos) return 0;
      // each remaining CBOR item consumes at least 1 byte
    bool err = false;
    size_t payload_count =
      get_payload_count(input+pos, header_size, &err);
    if (err) return false;
    if (payload_count > len − pos − expected) return false;
    expected = expected + payload_count;
  }
  return pos;
}
```

Fig. 6. C validator for raw CBOR bytes

This datatype extracts to C as a tagged union, with ref and narray extracting as C pointer types.

Then, using the copy writer combinators we defined in PulseParse, we implement a recursive serializer from values of this datatype.

The implementation of this example takes 150 lines of specification and 1000 lines of PulseParse implementation, which extract to around 800 lines of C code. No proof was necessary, since correctness and non-malleability are obtained by construction by virtue of typechecking the combinator calls. The full example is provided in EverParse[7].

B Constant-stack, arithmetically safe validation of raw CBOR bytes

Let `size_t validate_header(uint8_t *input, size_t len)` be a validator for CBOR item headers, returning the (nonzero) number of bytes consumed for a valid header, and 0 otherwise. Let

`size_t get_payload_count(uint8_t *input, size_t len, bool *err)`

be a function that takes a valid byte representation of a header and returns the number of expected CBOR items to validate in the payload; but sets `*err` to `true` if the expected length is greater than the length `len` of its input, to avoid any arithmetic overflow. Then, since the format of CBOR headers has the prefix property (the validity of a header does not change if any bytes are appended to it), the C function in Figure 6 `validate` is a memory safe, arithmetically safe, and functionally correct validator for CBOR items, returning the size of the valid CBOR item found at the beginning of the input buffer, or 0 if none:

C DICE Protection Environment

DICE Protection Environment (DPE) [Trusted Computing Group 2023] is a standard for a family of protocols to measure and cryptographically attest the integrity of the boot sequence of hardware ranging from IoT devices to cloud machines. DPE implementations support various *profiles*, exposing different interfaces and capabilities to clients. Ebner et al. [2025] provide a verified implementation of DPE in Pulse, supporting only the simplest profile, where a DPE client is expected to be executing in the same address space, sharing memory with the DPE attestation service. A more common profile instead allows a client to be dislocated from the DPE service, and for them to communicate over a transport using CBOR messages specified in CDDL.

The CDDL used in the DPE specification are all in a style that enable extension. For example, all messages are of the form `{ l => t, *(uint => any) }`, which, as explained in §4, is ambiguous. So, we adapt the specifications to add cuts, e.g., rewriting them to `{ l:t, *(uint => any)}`. Once in this form, EverCDDL proves the specifications unambiguous and generates Pulse code to parse and serialize CBOR formatted messages to and from typed data structures. In total there are four messages to the parsed as input to the DPE service and two messages that it serializes as output back to the client.

We adapt Ebner et al.'s DPE interface and add a layer on top of it that adds CDDL message parsing and serialization, with proofs in Pulse, demonstrating that the specifications yielded by EverCDDL are precise enough to express full functional correctness of application code manipulating CBOR messages in a given CDDL schema. For instance, our top-level specification of the sign API is shown below:

```
{ input ↦(p) i ** out ↦ _ } let ok = sign input out
{ if ok=Success then ( ∃ o sig tbs. input ↦(p) i ** out ↦ o **
     (is_tbs_bytes tbs i ∧ is_signature sig tbs ∧ is_serialized_sig o sig)
  ) else ... }
```

This triple states that with (fractional) ownership of an input buffer with bytes i and full ownership of an out buffer, if sign returns Success, then the input buffer is unchanged, the output buffer contains o, where o is a serialized signature sig of the to-be-signed bytes tbs from a well-formatted input buffer i. We also fully specify three possible modes of failure.

[7] See footnote 1