

VUPER: Verified ASN.1 UPER Parser

Xiaotian Zhou
xzz5503@psu.edu
The Pennsylvania State University
University Park, PA, United States

Kai Tu
kjt5562@psu.edu
The Pennsylvania State University
University Park, PA, United States

Ali Ranjbar
aranjbar@psu.edu
The Pennsylvania State University
University Park, PA, United States

Yilu Dong
yiludong@psu.edu
The Pennsylvania State University
University Park, PA, United States

Gang Tan
gtan@psu.edu
The Pennsylvania State University
University Park, PA, United States

Syed Rafiul Hussain
hussain1@psu.edu
The Pennsylvania State University
University Park, PA, United States

Abstract

ASN.1 is a widely used interface description language, and UPER (Unaligned Packed Encoding Rules) is one of its key encoding rules, particularly popular in security-critical domains such as cellular networks and vehicle-to-everything (V2X) communication. To ensure the correctness and security of this foundational infrastructure, we present VUPER, a framework for generating verified ASN.1 UPER parsers. We first formalize the notion of a bit-precise parser and identify properties that prove round-trip consistency for parsers and serializers, while accounting for ASN.1 features such as backward/forward compatibility. We then implement and verify parser and serializer combinators for ASN.1 basic types and structures, while adhering to the UPER specification. We also develop a compiler that translates ASN.1 definitions into verified parsers. Finally, we develop a dynamic testing framework using the VUPER parser as a test oracle. To empirically evaluate our approach, we test 7 open-source and 4 commercial ASN.1 parsers using 5G and V2X communication protocols. VUPER uncovers 20 types of inconsistencies in popular parsers and demonstrates stricter compliance with ASN.1 UPER standards. Additionally, we demonstrate concrete attacks by exploiting these parser vulnerabilities.

CCS Concepts

• Security and privacy → Formal methods and theory of security; • Networks → Protocol testing and verification.

Keywords

Binary Parser; Formal Verification; ASN.1

ACM Reference Format:

Xiaotian Zhou, Kai Tu, Ali Ranjbar, Yilu Dong, Gang Tan, and Syed Rafiul Hussain. 2026. VUPER: Verified ASN.1 UPER Parser. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3832737>

1 Introduction

Parsing and serializing binary data formats are fundamental tasks in network protocols. The parser must ensure the correct reconstruction of the original message encoded by the serializer and safely reject potentially malicious inputs. Interface description languages (IDL) streamline this process by decoupling parsing from the main program, thus allowing the reuse of parser libraries across protocols sharing the same IDL. Abstract Syntax Notation One (ASN.1) is a widely used IDL standardized by the International Telecommunication Union (ITU) [31]. Among ASN.1 encoding rules, such as Basic Encoding Rules or BER, Distinguished Encoding Rules or DER, Unaligned Packed Encoding Rules (UPER) are most widely used in security-critical wireless protocols such as cellular networks and vehicle-to-everything (V2X) systems due to their efficiency, compactness, expressiveness, and interoperability. Despite their prevalence and complexity, UPER implementations remain under-investigated [58] and lead to critical vulnerabilities, ranging from safety violations to subtle logic bugs [41].

One of the main reasons for such flaws is that the ASN.1 standard exists only as descriptive text, lacking a precise reference implementation. However, the intricate nature of ASN.1, including its support for various primitive types and combinators, ranges, optional fields, extensions, and the bit-level precision required by UPER, makes it very difficult to derive a fully correct implementation. This leaves the ASN.1 implementations highly error-prone, as evidenced by the over 100 reported CVEs [19]. Subtle flaws or non-conformance in the parser can also propagate issues throughout the downstream systems, potentially leading to authentication bypass [40], sensitive information exposure [38], or denial-of-service (DoS) [14] attacks.

Prior work on securing parsers has primarily focused on testing and fuzzing [28]. Although these approaches are effective for identifying vulnerabilities, they cannot provide the formal guarantees that eliminate entire classes of errors. In contrast, constructing parsers with provable security guarantees can achieve this by design [47, 62]. Such guarantees are typically based on a *round-trip* property [47, 62], which requires the parser and serializer to be mathematical inverses. This property ensures two critical principles: *non-ambiguity*, where a given encoding maps to at most one message, and *non-malleability*, where a message has only one canonical encoding. Enforcing these principles ensures that the parser correctly interprets the input while preventing all malformed ones.

Previous efforts on verifying binary formats cover both generic formats [22, 47, 60], and specific formats such as ASN.1 DER [42], and Protocol Buffer [64]. However, when it comes to ASN.1 UPER,



This work is licensed under a Creative Commons Attribution 4.0 International License. *CCS '26, The Hague, Netherlands.*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2871-6/2026/11

<https://doi.org/10.1145/3830454.3832737>

prior approaches have several critical limitations (summarized in Table 1). First, specifying the correctness property for UPER is complicated by the protocol's flexibility. For instance, EverParse [47] framework enforces a strict bijective mapping between parser and serializer, but this property cannot easily accommodate the extensibility requirements of UPER. Meanwhile, ignoring malleability [17, 60] entirely is overly permissive and risks overlooking malformed packets. Second, most frameworks are byte-oriented [18, 47, 62], creating a mismatch when applied to the bit-precise protocol UPER. Furthermore, high-performance zero-copy systems like EverParse [47] cannot be easily ported to UPER, because bit-precise protocols mandate explicit value extraction from arbitrary bit offsets. Adapting these architectures would require a complete redesign and likely negate their original performance advantages. Finally, most existing ASN.1 formalizations focus on DER [17, 21, 36, 42], whereas the only prior work on UPER [59] lacks support for the critical yet most common UPER features, such as CHOICE types and extensions while imposing a hard-coded limit (e.g., 37 bytes) on input.

To bridge these gaps, we present VUPER, the *first comprehensive* framework for representing bit-precise parsers and serializers and formally verifying the correctness of ASN.1 UPER parsers, using the Rocq prover [56]. VUPER achieves data compactness by using a bitstream model that disregards byte alignment, while still maintaining reasonable performance. VUPER guarantees a nuanced round-trip property between its parser and serializer to account for forward and backward compatibility. For messages without unknown extensions, a strict injection property ensures full non-malleability. Conversely, for messages with unknown extensions, the weak-injection property applies a more relaxed criterion: the parsed message can be successfully re-encoded, without insisting that the output exactly match the input. Together, these properties demonstrate that the parser rejects malformed or malicious inputs and remains correct with respect to the serializer. We also prove auxiliary properties essential for building other guarantees, including *local write* and *encode-consistency*. We adopt a *modular* parser-combinator approach. We develop a family of parsers and serializers that range from bit-level manipulation to structural types like CHOICE and SEQUENCE. Each component is formally verified to ensure round-trip consistency. By utilizing these combinators, we build a compiler that automatically transforms ASN.1 definitions into verified parser/serializer pairs.

We further develop a testing framework using our formally verified VUPER parser (extracted from Rocq to OCaml) as a trusted oracle. The framework helps systematically uncover discrepancies in other ASN.1 implementations. To effectively generate testing inputs, we use grammar-aware mutation by incorporating ASNFuzzGen [14] into the fuzzing loop. To ensure precise comparison, we utilize the JSON Encoding Rules (JER) [33] to normalize and compare the values produced by each parser.

We evaluated 11 parser libraries by testing them with uplink and downlink messages from 5G and Intelligent Transport Systems (ITS) protocols while using the verified parser as the oracle. Our evaluation includes 7 open-source and 4 closed-source implementations, notably two ASN.1 UPER parsers from commercial 4G and 5G basebands. We find that VUPER enforces stricter compliance than most existing implementations and uncovers 20 types of inconsistencies in other libraries, including lenient constraint checks, value

corruption, and deviations from the UPER standard. To further evaluate the security implications of these defects, we demonstrate 4 concrete attacks against two widely used open-source 4G/5G stacks. These exploits include DoS via unfiltered inputs and cryptographic key misconfigurations due to parser misinterpretation. We also benchmark the performance of the VUPER-generated parser on 5G messages, which is on average 4 times slower than C libraries [3, 5], and 4 times faster than Python libraries [7]. This is expected as VUPER prioritizes formal correctness over efficiency and is intended to be used as a reference implementation for testing other ASN.1 parsers and fixing bugs. This is a key first step toward a performant, low-overhead, and formally verified ASN.1 UPER parser.

Contributions. We make the following contributions.

- We design VUPER, an automated, bit-precise, and correct-by-construction ASN.1 UPER parser and serializer library.
- We prove and implement the *round-trip* properties for extensions, and the basic and ASN.1 parser combinators of our parser.
- We develop a testing framework based on VUPER as a trusted oracle to effectively validate existing UPER parsers by leveraging VUPER's formal guarantees.
- We evaluate 7 open-source and 4 commercial parser implementations and uncover 20 types of non-compliance, which can lead to sensitive information exposure and DoS.

2 Background

Abstract Syntax Notation One (ASN.1) is a formal language for defining the vendor-neutral data structures used in telecommunication protocols [23, 25, 30]. Its primary strength lies in decoupling abstract data logic from its physical bit/byte representation, ensuring interoperability across diverse systems. ASN.1 provides a highly expressive syntax for this purpose. Figure 1b shows an ASN.1 definition of PDSCH-ServingCellConfig from the 5G Radio Resource Control (RRC) protocol [10]. The message is transmitted from 5G base stations to user equipment (UE), and contains parameters necessary for processing the Physical Downlink Shared Channel (PDSCH) on the corresponding cell. The message is a SEQUENCE (i.e., struct) of fields and contains primitive types like BOOLEAN and INTEGER and user-defined types like ServCellIndex. It also includes parameterized types and markers (e.g., OPTIONAL) for fields to describe their behavior.

Extensions. The example in Figure 1 also illustrates ASN.1 extensions. The definition in Figure 1a shows the base message from an earlier 5G release, future-proofed with the '...' extension marker. The definition in Figure 1b is a later release with new capabilities and adds new fields such as maxMIMO-Layers. These additions are organized into ExtensionAdditionGroups, wrapped by the double square brackets ([...]). Extensions are essential for ensuring interoperability, enabling systems operating on different releases to communicate effectively without errors. For forward compatibility, if an old system (e.g., Rel-15.2.1) receives a message like Figure 2a from a newer system (e.g., Rel-16.3.1), it would process known fields such as nrofHARQ, while skipping the unknown fields like maxMIMO-Layers, resulting in the decoded messages in Figure 2b. Similarly, backward compatibility means the newer system should handle the absence of certain fields gracefully. Furthermore, ASN.1

```

PDSCH-ServingCellConfig ::= SEQUENCE {
  codeBlockGrpTx      SetupRelease { PDSCH-CodeBlockGrpTx } OPTIONAL,
  xOverhead           ENUMERATED { x0h6, x0h12, x0h18 } OPTIONAL,
  nrofHARQ            ENUMERATED { n2, n4, n6, n10, n12, n16 } OPTIONAL,
  pucch-Cell         ServCellIndex OPTIONAL,
  ... }

(a) Definition in RRC Release 15.2.1

PDSCH-ServingCellConfig ::= SEQUENCE {
  codeBlockGrpTx      SetupRelease { PDSCH-CodeBlockGrpTx } OPTIONAL,
  xOverhead           ENUMERATED { x0h6, x0h12, x0h18 } OPTIONAL,
  nrofHARQ            ENUMERATED { n2, n4, n6, n10, n12, n16 } OPTIONAL,
  pucch-Cell         ServCellIndex OPTIONAL,
  ...,
  [[ maxMIMO-Layers    INTEGER (1..8)                OPTIONAL,
   processingType2Enabled BOOLEAN                     OPTIONAL ]],
  [[ pdsch-CodeBlockGrpTxList-r16
     SetupRelease { PDSCH-CodeBlockGrpTxList-r16 } OPTIONAL
  ]] }

(b) Definition in RRC Release 16.3.1

```

Figure 1: Definition of PDSCH-ServingCellConfig in two different releases with some field and type names simplified.

```

tmobile-PDSCH-ServingCellConfig ::= {
  nrofHARQ      n16,
  maxMIMO-Layers 2
}

tmobile-PDSCH-ServingCellConfig ::= {
  nrofHARQ      n16
  -- Skipped unknown field
}

```

(a) Message from Newer Version (Rel 16.3.1) **(b) Message After Decoded by Older System (Rel 15.2.1)**

Bitmap for Optional	# of Extension Groups	Length field For Open Type	Encoding of 2	
1	0010	101	00000001	10
00000001	10	00000001	10	001
000				

Extension Existence Indicator Encoding of n16 Bitmap for Extension Bitmap for Extension Group Optional Fields Padding

(c) Bit Encoding of Message in (a)

Figure 2: Concrete PDSCH-ServingCellConfig Message (Rel. 16.3.1) sent by T-Mobile Network with its Bit Encoding

extensions must not be confused with messages labeled as ‘extension’ or other protocol-specific extension mechanisms. To highlight the essential role of extensions in complex networks, there are more than 500 occurrences of extension markers out of a total of 2,500 defined message types in 5G RRC (Rel 17) [10].

UPER (Unaligned Packed Encoding Rules). Once the abstract type is defined, the value for each message can be physically represented using one of several mutually incompatible encoding rules, each suited for different purposes.

Compact Encoding. Rules of BER/DER use a byte-aligned tag-length-value (TLV) format. In contrast, Packed Encoding Rules (PER) prioritize compactness, with UPER going furthest by eliminating all byte alignment. For example, a constrained type like INTEGER(1..8) is encoded in UPER using only 3 bits instead of 4 bytes (32-bit int) to represent values from 1 to 8. Such a setup is perfect for bandwidth-limited environments. For instance, the RRC protocols in 4G/5G [10] use UPER for efficient over-the-air messaging. UPER is likewise employed in Intelligent Transport Systems (ITS) standards [23, 49], encoding messages from vehicle-to-vehicle communication to traffic management.

Canonical vs. Basic UPER. Canonical UPER enforces a mandatory 1-1 correspondence between the value and its bit representation, while Basic is lenient on some occasions.

Illustration of UPER’s bit-level encoding. Figure 2c illustrates the bit-level encoding of a concrete PDSCH-ServingCellConfig message defined in Figure 2a. The encoding begins with a one-bit flag to indicate the presence of extensions, followed by a presence bitmap for the optional fields for the *sequence root* (items before the extension marker). In this message, the bitmap is 0010 because only the 3rd field out of the 4, nrofHARQ, is present. The encoded values of all present fields appear sequentially after the bitmap. Next is the *extension encoding*, which starts with a length determinant, a binary value indicating the number of extension groups. In this case, the value 0000001 represents 2 extension groups (the count starts at 1 in this encoding scheme). This is followed by a bitmap indicating which extension groups are present. Here, since only the first group exists, the bitmap is 10. Each group carries its own presence bitmap and field values. Here, the first extension group only includes the first field maxMIMO-Layers out of the two, so its bitmap is 10. The value of that field follows. Finally, the extension group is wrapped in an Open Type field. During encoding of an Open Type field, the inner type is first generated and padded to the nearest octet boundary. A length determinant is then prefixed to specify the total number of octets, followed by the octet-aligned payload. This structure allows a parser to safely skip unrecognized extensions by advancing the bitstream by the indicated octet length, which facilitates forward compatibility.

3 Design Principles of VUPER

To present the design principles of VUPER, we first begin with the parser’s round-trip properties, then turn to our verification approaches, and finally discuss the technical challenges.

3.1 Properties of Parsers

In general, parser security is often framed in terms of two dual properties: *surjection* and *injection* [47, 59, 62]. A serializer $s : A \rightarrow \mathbb{B}$ maps data of type A into its encoded form $\mathbb{B}$, usually a byte array. And a parser $p : \mathbb{B} \rightarrow A \cup \perp$ does the reverse, converting an encoded input back to the value in A , or returning $\perp$ for invalid inputs.

Surjection. A parser p is a surjective inverse of a serializer s if, for any message $a \in A$, it satisfies $p(s(a)) = a$. Surjection also ensures non-ambiguity: no two distinct messages share the same encoding. Formally, $\forall a, a' \in A$, if $s(a) = s(a')$, then $a = a'$.

Injection. It requires that the serializer recovers exactly the original encoding whenever parsing succeeds. Formulated as $\forall b \in \mathbb{B}$, $s(p(b)) = b$ or $p(b) = \perp$. This implies non-malleability: for all $b, b' \in \mathbb{B}$, if $p(b) = p(b')$, then either $b = b'$ or $p(b) = \perp$, ensuring each valid encoding corresponds to a unique message.

These properties align with Canonical UPER, which enforces a strict bijective mapping of values and bitstrings. The *surjection* property ensures that every encoded value can be correctly parsed. This property establishes the *completeness* of the parser relative to the serializer, ensuring it never rejects correctly encoded inputs. The *injection* property can prevent security issues by properly filtering all malformed inputs [47, 62], such as out-of-range values or non-canonical encodings. *Termination* is also crucial for program

reliability by preventing infinite loops that may cause freezes or resource exhaustion. We exclude side-channel security [18] as enforcing constant-time execution is generally impractical for ASN.1 UPER, where payloads range from simple values to complex nested structures [10]. We also exclude input exhaustion, as trailing bits do not affect the semantic correctness of a successfully decoded value.

Threat Model. Since ASN.1 parsers are used in many network protocols, a remote attacker could exploit parser vulnerabilities by transmitting specially crafted payloads to the victim’s network interface. For instance, the RRC protocol [10, 25] between the UE and gNB exposes 3 possible attack scenarios. ❶ *A fake base station* can lure the UE to connect and send malicious RRC messages to the UE, causing DoS of the user [44, 50, 57]. ❷ On the other hand, an *attacker-controlled user device* can also send malformed messages to the base station to which it connects [29, 34]. ❸ RRC messages *transmitted between base stations* (e.g., during handovers) are also vulnerable. Studies [51, 63] show that femtocells (compact indoor base stations) can be compromised to exploit this interface.

Table 1: Comparison with existing verified parsers

System	Format	Properties			Features			
		Surj	Inj	Weak-Inj	Bit	Ext	Auto	Fast
VUPER	ASN.1 UPER	●	●	●	●	●	●	○
EverParse [47]	Custom DSL	●	●	○	○	-	●	●
Narcissus [22]	Custom DSL	●	●	●	●	-	○	○
ASN1* [42]	ASN.1 DER	●	●	○	○	○	●	○
ARMOR [21]	ASN.1 DER	●	●	○	○	○	●	○
V2V [59]	ASN.1 UPER	●	●	○	●	○	●	●
ASN1SCC [17]	ASN.1/ACN	●	○	○	●	○	●	○

●: full support, ○: no support, ●: limited support, -: not applicable; Ext: ASN.1 Extension; Auto: automatically generating a verified parser; Fast: efficient implementation. **Surjection**; **Injection**

3.2 Challenges and Motivation of VUPER

To motivate the design of VUPER, we analyze the unique challenges UPER poses for formal specification and verification, and reasons why existing verified parsing frameworks are insufficient for handling its complexities (Table 1).

C1: Backward/Forward Compatibility with Extensions. As discussed in §2, ASN.1 extensions are fundamental to the evolution of 4G, 5G, and V2X protocols. This brings additional challenges because extensions would break the *injection* property, as they require the parser to accept messages from different versions. For example, in Figure 2, when the old system receives the Message 2a and attempts to re-encode it, the output is 0 0010 101. This completely ignores all extensions (since none are present at Rel 15.2.1) and results in an encoding that differs from Figure 2c. Similarly, it is not a one-to-many relation between the value and the bit representation either, because while the parser is required to accept these unknown extensions, the serializer is strictly limited to the fields defined in its own version of the standard.

This makes adopting strict, non-malleable frameworks like Everparse [47] or Compars [62] difficult. There are two potential workarounds, but they are both flawed. The first is to reject all messages containing unknown extensions, but this severely hinders interoperability in real networks where extensions are common. The second is to expand the input space to include all unknown extensions for the encoder, but this undermines the encoding model.

Using a fully malleable format, such as Narcissus [22], is also not appropriate since ASN.1 UPER is fundamentally not a malleable encoding. Thus, our approach will extend the round-trip theory.

C2: Bit-level Precision in a Majority Byte-Aligned World.

UPER is a bit-precise encoding format that operates at the level of individual bits rather than bytes, making implementation and formal verification more complex. Decoding UPER requires maintaining a precise bit cursor and performing complex bitwise manipulation like shifting and masking to reconstruct parsed fields.

While it is possible to represent bit-level protocols in some systems like Narcissus [22], their optimized extraction mechanisms only work with byte-aligned formats. Using these tools to handle bit-level granularity would incur significant performance penalties. Meanwhile, most other parser frameworks operate on byte-aligned formats, because common protocols like ASN.1 DER are byte-aligned. Thus, approaches like ASN1* [42] and EverParse [47] cannot be easily ported to UPER. This is because they use high-level parsers/serializers and low-level accessors/readers to track byte positions for zero-copy implementation. Adapting these systems to bit-level formats would require significant rewriting of both high- and low-level components to handle bit-level positions and may incur performance overhead. And their zero-copy feature offers minimal benefits for UPER since values at arbitrary bit positions cannot be dereferenced and still need bit-level extraction.

C3: Explicit and Implicit Constraints. Constraints are essential for efficient serialization in UPER, but complicate verification. Explicit constraints, such as INTEGER (1..8) (Figure 1b), enforce fixed value ranges that are easily formalized. On the other hand, implicit constraints during encoding add more complexity. For example, encoding an INTEGER requires converting the value to signed two’s-complement binary, normalizing it to the most compact octet-aligned form, and prefixing it with a length determinant, while always using the minimal number of octets. Similar challenges arise with structural types like SEQUENCE, where fields of non-structural type with DEFAULT values must be omitted if their value is equal to the default.

C4: Encoding Open Type Fields. Encoding ASN.1 Open Type Fields is nuanced. Introduced in §2, these fields are prefixed with their length, which allows older parsers to skip unknown extensions (Figure 2). This creates a circular dependency where the number of bits used for the length prefix itself depends on the total length of the value, which is only known after encoding is complete. Consequently, attempting to integrate these fields into verified parser frameworks like EverParse [47] is non-trivial, as it requires dynamic memory allocation and multi-pass encoding for each Open Type encounter. Our approach calculates the length beforehand by keeping a separate length function, eliminating the need for a temporary buffer and copy operation. However, this introduces additional complexity, as the length function must be verified to be consistent with the encoder to ensure correctness. Furthermore, Open Type fields impose a special fragmentation process when the length exceeds 16K bytes. But this occurs very rarely in practice [10, 25]; thus, we limit the possible length for an Open Type encoding to 16K bytes (128K bits) for simplicity. While this limit is not an issue in practice, it has consequences for formal verification. Within the logic of the theorem prover, this constraint restricts the

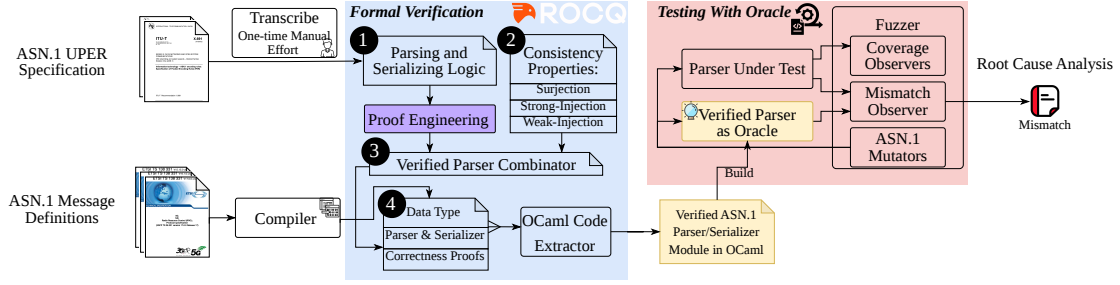


Figure 3: Architecture Overview

domain of possible messages for the parser and serializer, which complicates how we define certain correctness properties.

3.3 Design Philosophy

Objective & Scope. Our objective is to formally verify the correctness of ASN.1 parser and serializer implementations. This practical focus on implementation distinguishes our work from efforts like ASN1* [42], which develops a formal model of the ASN.1 language itself, treating verified DER parsers as executable semantics to ensure that any valid ASN.1 structure produces a verified parser. We therefore exclude a verified compiler: since ASN.1 semantics are defined by the encoders and decoders the compiler produces [52], verifying it is a tautological exercise of marginal value.

We use the Rocq prover [56] to implement the verified parser and serializers. Termination is ensured by Rocq’s type checker, as all recursive functions are defined by structural recursion over standard inductive types. We do not explicitly prove memory safety, but rely on the memory safety provided by the OCaml runtime, and trust Rocq’s extraction mechanism to preserve the properties established in the proof assistant.

Extended Round-trip. We choose to verify the parser’s correctness by proving an extended round-trip consistency with the corresponding serializer. In contrast, previous approaches such as Narcissus [22] and ARMOR [21] verify both the parser and the serializer against a single source of truth: a formal specification defined as a relation between a value and its encoding. This approach offers advantages in settings like Protocol Buffers [64], where one-to-many encodings must be supported. However, UPER does not involve one-to-many encodings as previously discussed (§3.2). Moreover, these approaches ultimately provide mathematically equivalent guarantees but require additional proof overhead to show that the parser conforms to the bijective format.

3.4 VUPER Workflow

Figure 3 illustrates the overall architecture of VUPER. We ❶ transcribe the ASN.1 UPER specification [32] into parsing and serializing logic in Rocq, and ❷ formalize the concept of round-trip consistency properties (§4.3). Next, we develop a set of verified basic combinators (§4.4.1), which serve as the foundation for ASN.1 primitive types. Subsequently, we construct structural combinators (§4.4.2), like SEQUENCE, and prove their correctness. Together,

these form ❸ our library of verified parser combinators. We implement a compiler (§4.5) which generates ❹ data types and verified parsers/serializers from any ASN.1 definition by leveraging our library of verified combinators. One key benefit of a verified parser is that it can effectively work as a trusted *test oracle* (§5), enabling rigorous conformance testing of other parsers.

4 Design Details of VUPER

We now discuss the design details of VUPER, including the parser/serializer definition, solution to the challenges, correctness guarantees, verified combinators and compiler.

4.1 VUPER Serializer and Parser Definitions

Our framework is built on parsers and serializers that operate directly on a byte array $\mathbb{B}$ (where a byte is $\{0, \dots, 255\}$), which we treat as a continuous bitstream. To navigate this stream, we use a precise bit position indicator, $\mathbb{P} := \mathbb{N} \times \{0, \dots, 7\}$, representing a byte index and a bit offset within that byte. Let $\text{Flg} := \text{SameVer} | \text{DiffVer}$ to indicate whether parsing results in the same version or a different version. To enforce type constraints, these parser/serializer functions are parameterized by a message type A and a refinement function $P : A \rightarrow \text{Prop}$. This refinement maps a value of type A to a logical proposition defining its constraint. For example, the constraint for `maxMIMO-Layers` in Figure 1b is `INTEGER (1..8)`, so its refinement is $\text{fun } (x : \mathbb{Z}) \Rightarrow 1 \leq x \leq 8$. We define their type in the curried form as follows.

$\text{Parse } A \ P := \mathbb{B} \rightarrow \mathbb{P} \rightarrow \text{option}(\{x : A \mid P x\} \times \mathbb{P} \times \text{Flg})$

$\text{Serialize } A \ P := \mathbb{B} \rightarrow \mathbb{P} \rightarrow \forall a : A, \text{option}(\mathbb{B} \times \mathbb{P} \times P a)$

In essence, the parser takes a buffer and a starting bit position. Upon success, it returns a tuple containing: (1) the sigma type of the parsed value, along with a proof that it satisfies the condition P ; (2) the new bit position after the read; and (3) the versioning flag. The serializer takes a buffer, a starting position, and a value to write. When successful, it returns the updated buffer, the new bit position along with a proof that the value satisfies the required condition P . Both functions are wrapped in an option type to handle errors, such as reaching the end of the buffer unexpectedly or encountering a value that violates its expected constraints.

4.2 Solutions to Challenges of Verifying UPER

Weak-Injection Property. To solve challenge C1 (forward/backward compatibility), we adopt a flag-based version detection mechanism and a corresponding *Weak-Injection* property [55], formally defined in §4.3. The parser sets a flag if it encounters an unexpected number of extensions, signaling a version mismatch. This lightweight approach extends security guarantees with minimal overhead. Consequently, our correctness guarantees are two-tiered: the standard *Injection* property applies when versions match, while the *Weak-Injection* property applies in all cases. *Weak-Injection* ensures that the serializer can handle any output from the parser, guaranteeing that even when re-encoding does not produce an identical bitstring, the resulting value remains valid and constrained rather than arbitrary. The trade-off is the proof effort to establish this property for all extensible datatypes.

Bitstream Approach. To better handle Challenge C2 (bit-level precision), we utilize direct, efficient bit-level manipulation. There can be some alternatives. One potential way is to map the input stream to a list bool in Rocq. While this approach seems easier in theory, it can be highly inefficient, especially when the ASN.1 messages can be several thousand bits long in complex network protocols (e.g., 4G, 5G). Another alternative involves encoding fields into independent byte buffers and subsequently concatenating them. However, this process is inefficient for nested structures, such as SEQUENCE, where each field might be realigned to maintain bit-level correctness and concatenated multiple times. In contrast, VUPER operates directly on a bitstream while tracking the current bit offset, naturally supporting arbitrary bit alignment without requiring repeated shifting or copying. We reformulate the consistency properties for our parser and serializer definitions without compromising the level of assurance, a non-trivial task discussed in §4.3.

Parser Combinators. For Challenge C3 (explicit & implicit constraints), we use parser combinators to modularly construct data types, parser/serializers, and proofs, tackling ASN.1's complexities and enabling scalability of the verification process. For explicit constraints, we employ combinators like *restrict* to layer value constraints over other parsers. For implicit constraints, we compose basic verified parser combinators, but the process itself is burdened with proofs. To illustrate, supporting the INTEGER type requires defining a *mapping* between the integer and a dependent pair of a length determinant, and the integer's two's complement form. The first task is to restrict the dependent pair to only the minimum representation. Next, we must prove that this mapping constitutes a bijection to maintain the *round-trip* properties. This proof takes approximately 800 lines of code (LoC) because Rocq employs abstract mathematical representations of integers or natural numbers rather than machine integers, making seemingly simple procedures surprisingly complex. Finally, we design ASN.1-specific combinators to handle complex structural logic, as well as addressing C2. Details of combinators are in §4.4.

Open Type Field. For Challenge C4, we define a separate length function with type $A \rightarrow \text{option } \mathbb{N}$, where the option type handles inputs violating specific constraints. Furthermore, restricting the length of Open Types complicates the *weak-injection* property, as additional constraints are required to guarantee the production of valid encodings.

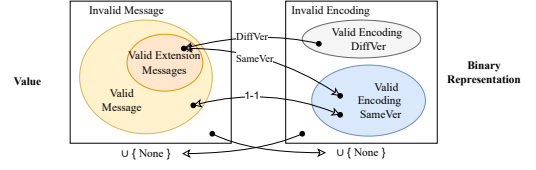


Figure 4: Mapping between values and their binary encoding.

4.3 Correctness Guarantees

Figure 4 illustrates the property we aim to prove. We define the surjection and injection properties first. We assume that $A : \text{Set}$, $P : A \rightarrow \text{Prop}$, $\text{enc} : \text{Serialize } A \ P$, $\text{dec} : \text{Parse } A \ P$, and the length helper function $\text{msg_len} : A \rightarrow \text{option } \mathbb{N}$. Also, we assume *proof irrelevance* in all definitions. The properties are summarized in Table 2.

Surjection and Strong-Injection. We can directly embed the mathematical properties of surjection and injection (§3.1) into the definitions of our serializer and parser. For *surjection*, we formally define the property as follows.

Definition *surjection* := $\forall (b \ b' : \mathbb{B}) (p \ p' : \mathbb{P}) (a : A) (pf : P \ a)$,
 $\text{enc } b \ p \ a = \text{Some } (b', p', pf) \rightarrow$
 $\text{dec } b' \ p = \text{Some } (\text{exist } P \ a \ pf, p', \text{SameVer})$.

This definition asserts that if encoding a value a from position p successfully yields a buffer b' and position p' , decoding b' from the same starting position p must return the same a , and position p' .

Meanwhile, injection requires that if a buffer b decodes to some value a , then re-encoding a should yield the same buffer. However, equivalence between buffers must be carefully defined, as re-encoding creates a new buffer, with potentially different positions. To address this, we introduce the notion of *equiv_bits*, which captures bitwise equivalence between two buffer slices. Its formal type is: $\text{equiv_bits } (b_1 \ b_2 : \mathbb{B}) (p_1 \ p'_1 \ p_2 \ p'_2 : \mathbb{P}) : \text{Prop}$. Intuitively, the slice of buffer b_1 in the window $[p_1, p'_1)$ is identical, bit for bit, to the slice $[p_2, p'_2)$ of buffer b_2 . Additionally, *equiv_bits* implies that $p_1 \leq p'_1 \leq \text{buf_len}(b_1)$ to ensure valid buffer bounds, where buf_len returns the bit-length of the buffer. Moreover, to avoid rejection due to insufficient buffer length, we quantify over any buffer with the same length as the original. We define *strong injection*.

Definition *strong_injection* := $\forall (b : \mathbb{B}) (p \ p' : \mathbb{P}) (a : A) (pf : P \ a)$,
 $\text{dec } b \ p = \text{Some } (\text{exist } P \ a \ pf, p', \text{SameVer}) \rightarrow$
 $\forall (b_0 : \mathbb{B}), \text{buf_len } b_0 = \text{buf_len } b \rightarrow$
 $\exists (b_1 : \mathbb{B}), \text{enc } b_0 \ p \ a = \text{Some } (b_1, p', pf) \wedge \text{equiv_bits } b \ b_1 \ p \ p' \ p \ p'$.

However, such straightforward transcription would not suffice. The defined *strong-injection* properties do not imply *non-malleability*.

Definition *non_malleability* :=
 $\forall (b_1 \ b_2 : \mathbb{B}) (p_1 \ p'_1 \ p_2 \ p'_2 : \mathbb{P}) (a : A) (pf : P \ a)$
 $\text{dec } b_1 \ p_1 = \text{Some } (\text{exist } P \ a \ pf, p'_1, \text{SameVer}) \rightarrow$
 $\text{dec } b_2 \ p_2 = \text{Some } (\text{exist } P \ a \ pf, p'_2, \text{SameVer}) \rightarrow$
 $\text{equiv_bits } b_1 \ b_2 \ p_1 \ p'_1 \ p_2 \ p'_2$.

This is due to the potential dependency of parsers on external factors like bit position. For instance, consider two non-malleable serializer-parser pairs, $(\text{enc}_1, \text{dec}_1)$ and $(\text{enc}_2, \text{dec}_2)$. We can construct a third serializer-parser pair that uses $(\text{enc}_1, \text{dec}_1)$ if the bit position $\leq (1, 0)$ and uses $(\text{enc}_2, \text{dec}_2)$ otherwise. The third pair

satisfies this version of *strong injection* property, yet it clearly fails to produce a unique encoding. Therefore, we need to ensure the encoder is independent of both the given buffer and write position.

Definition `encode_invariance` :=

$$\forall (b_1 \ b'_1 \ b_2 \ b'_2 : \mathbb{B}) (p_1 \ p'_1 \ p_2 \ p'_2 : \mathbb{P}) (a : A) (pf : P \ a), \\ \text{enc } b_1 \ p_1 \ a = \text{Some } (b'_1, p'_1, pf) \rightarrow \text{enc } b_2 \ p_2 \ a = \text{Some } (b'_2, p'_2, pf) \rightarrow \\ \text{equiv_bits } b'_1 \ b'_2 \ p_1 \ p'_1 \ p_2 \ p'_2.$$

This property ensures the serializer produces a canonical bit-level representation for any value, preventing context-dependent outputs. Thus, we can prove that a combination of *strong-injection* and *encode-invariance* implies *non-malleability*. Similarly, *surjection* does not imply *non-ambiguity* automatically. We define the dual property *decode-invariance*, to ensure that the parser is not dependent on unrelated factors like position or buffer content.

Local Write. One other important property is *local write*. It ensures that encoding into a buffer b from position p to p' does not modify bits outside this range. Here, $0 := (0, 0)$ indicates the starting position of a buffer, while $b.\text{end}$ indicates the end.

Definition `local_write` := $\forall (b \ b' : \mathbb{B}) (p \ p' : \mathbb{P}) (a : A) (pf : P \ a),$

$$\text{enc } b \ p \ a = \text{Some } (b', p', pf) \rightarrow \\ \text{equiv_bits } b \ b' \ 0 \ p \ 0 \ p \wedge \text{equiv_bits } b \ b' \ p' \ b.\text{end} \ p' \ b' \ b.\text{end} \wedge p \leq p'.$$

This property is crucial for composing serializers and parsers, such as in a format that concatenates encoded types A and B . This property prevents the serialization of B from corrupting the previously-written bit segments of A . With it, we can confidently chain proofs, applying A 's surjection proof to its segment without interference from B . Additionally, *local write* implies that the buffer length remains unchanged between b and b' , a useful implication in establishing *strong-injection*.

Weak-Injection Property. This property aims to prevent malformed packets while maintaining flexibility.

Definition `weak_injection` :=

$$\forall (b : \mathbb{B}) (p \ p' : \mathbb{P}) (a : A) (pf : P \ a) (\text{flg} : \text{Flg}), \\ \text{dec } b \ p = \text{Some } (\text{exist } P \ a \ pf, p', \text{flg}) \rightarrow \\ \exists l_{\min} : \text{nat}, \forall (p_0 : \mathbb{P}) (b_0 : \mathbb{B}), \\ \text{to_nat } p_0 + l_{\min} \leq \text{buf_len } b_0 \rightarrow \text{msg_len } a \leq' 128K \rightarrow \\ \exists (b'_0 : \mathbb{B}) (p'_0 : \mathbb{P}), \text{enc } b_0 \ p_0 \ a = \text{Some } (b'_0, p'_0, pf) \wedge \\ \text{to_nat } p'_0 \leq \text{to_nat } p_0 + l_{\min}.$$

The definition establishes that if we can decode some buffer b to value a , then a can be encoded into some buffer b' , provided sufficient space. An additional constraint applies to the length of an encoding of a , without which, the length limit on Open Type field cannot guarantee the success of the encoding procedure at the end.

Finally, the *length-consistency* property ensures the length function `msg_len` accurately computes the encoding length as the difference between final and initial positions. This answers the Open Type field issue in Challenge C4, and aids the proofs of properties like *surjection* for Open Type fields.

4.4 Verified Parser Combinators

With correctness guarantees defined, we next discuss our combinator construction and proof engineering processes. We first introduce a notation for `Fmt` ($A : \text{Set}$) ($P : A \rightarrow \text{Prop}$) as a record that encapsulates the parser, serializer, and associated round-trip consistency proofs (the extended version [66]). This restructuring

Table 2: Correctness properties

Property	Description
<i>Surjection</i>	Preventing miscommunication
<i>Strong-Injection</i>	Rejecting all non-canonical packets
<i>Weak-Injection</i>	Prevent malformed values
<i>Local Write</i>	Preventing unintended writes to the buffer
<i>Encode-Invariance</i>	Ensuring the serializer is buffer or position-independent.
<i>Decode-Invariance</i>	Ensuring the parser is buffer- or position-independent.
<i>Length Consistency</i>	Crucial for verifying Open Type Field serializer correctness.

does not affect semantics or guarantees but improves clarity and facilitates combinator use.

4.4.1 Basic Parser Combinators. Basic Bit n Format. While not strictly a combinator, it acts as the direct interface function between the byte array $\mathbb{B}$ and the data type `nat`, enabling basic bit-level read/write operations, and works as the cornerstone of all other parsers and serializers. The decoder loads a byte sequence into a native integer to extract bit-fields using bitwise shift and mask operations. The encoder employs a read-modify-write process to maintain *local write*, ensuring unrelated bits are not accidentally modified. The target bytes are loaded into an integer, modified only at the precise bit offset, and then written back to the buffer. These proofs are complex, requiring precise, low-level reasoning about bitwise manipulations and bijective bytes-to-integer conversions.

Basic Combinators. To construct parsers for basic types like `BOOLEAN` or `OCTET STRING` from our primitive bit parser, we use a set of format combinators. The *map* combinator, for instance, transforms an existing format into a new data type with the same binary representation. Consequently, a `BOOLEAN` format can be efficiently modeled using the *map* and a basic bit format. Basic combinators can also propagate the metadata `Flag`.

Table 3: Selected combinators (refinement omitted)

Combinator	Input (Correctness Rules) $\rightarrow$ Output	Description
<i>basic-bit n</i>	<code>Fmt $\mathbb{N}$</code>	Formats a natural number between $[0, 2^n)$ as an n -bit integer.
<i>map</i>	$s_A : \text{Fmt } A, \text{ and bijective map between } A \text{ and } B \rightarrow \text{Fmt } B$	Formats type B by applying the bijective function to the existing format of A
<i>dependent append</i>	$s_A : \text{Fmt } A, \text{ and } s_B : A \rightarrow \text{Fmt } B \rightarrow \text{Fmt } (A \times B)$	Formats a pair (a, b) where the format for the second element b depends on the value of the first element a
<i>list</i>	$s_A : \text{Fmt } A, n : \mathbb{N} \rightarrow \text{Fmt } (\text{list } A)$	Formats a fixed-length list of n elements, each of type A .
<i>CHOICE</i>	$\forall i \in 1..n, s_i : \text{Fmt } A_i \rightarrow \text{Fmt } (\text{nth } A_i)$	Formats a tagged union that represents a value that can be one of several predefined types
<i>SEQUENCE</i>	$\forall i \in 1..n, s_i : \text{Fmt } A_i, \text{ mod}_i \in \{\text{Opt}, \text{Dft}, \text{Nor}\} \rightarrow \text{Fmt } (A'_1 \times \dots \times (A'_n \times \tau))$	Formats an ordered collection of items of different types, with modifiers for each item

4.4.2 ASN.1 Combinators. While general-purpose combinators are versatile enough to model ASN.1 combinator types like `SEQUENCE-OF` via *dependent append* and *list* format, they struggle with the intricate requirements of extensible `SEQUENCE` types. First, constructing a `SEQUENCE` using generic combinators like *dependent-pair* and *map* leads to verbose definitions, especially when handling inter-field dependencies, such as the optional-field bitmap. More importantly, the basic parser combinators are designed to be bijective by default, but the `SEQUENCE` parser needs to identify and skip unknown extensions. It is natural to leverage the structured nature of ASN.1 by introducing a set of specialized, ASN.1 format

combinators for types such as CHOICE and SEQUENCE, which correctly handle their complex extension mechanisms.

Sequence. The SEQUENCE combinator maps an ordered list of fields, marked as normal, OPTIONAL, or DEFAULT, to a right-nested tuple (shown in Table 3). Field types are adjusted based on their modifiers, e.g., converting OPTIONAL fields to option types. However, proving round-trip consistency is not straightforward because the OPTIONAL field bitmap creates a non-local dependency between presence bits at the start of the SEQUENCE and field data later in the stream. Our solution is to prove the components separately. First, we prove the round-trip consistency of the bitmap. Second, we inductively prove field consistency by parameterizing the parser with the decoded bitmap. Finally, we use a prepend correctness property to formally link these two separate proofs, guaranteeing the correctness of the entire SEQUENCE encoding.

Sequence Extensions. Parsing extensions is complicated due to the potential mismatch between the number of decoded extensions (n) and the number expected by the current version (m). While weak-injection guarantees a valid encoding for the length of n , it fails to establish the validity of the count m 's encoding. We thus define a valid-encoding property ensuring that any value satisfying a specific refinement condition has a valid encoding. We prove this property for the length determinant and the list combinator to validate the encoding of length m and the extension bitmap.

4.5 Compiler

To handle the scale of real-world protocols like 5G, we develop an automated compiler that translates ASN.1 definitions into verified Rocq code. The compiler recursively generates types and formats using the verified combinators. It also aims to provide an intuitive and user-friendly interface, for example, mapping SEQUENCE to a Record type instead of a nested product. The compiler also generates helpful lemmas and proof obligations. For example, it may establish a one-to-one mapping between an ENUMERATED type and a constrained integer with the proofs generated using custom tactics. More details are discussed in the extended version [66].

5 Testing with Verified Oracle

VUPER's formal guarantees make it a unique trusted oracle for testing. By comparing other parsers against it, we can detect subtle implementation flaws. This approach is particularly effective at uncovering bugs common to all implementations, which is a critical blind spot for traditional differential testing [12]. This process also serves a dual purpose as it provides crucial empirical validation for VUPER itself. Formal verification is only as strong as its underlying specification. By testing against widely-used systems, we confirm that our interpretation of the natural-language ASN.1 standard is correct and ensure real-world interoperability. Any discrepancies can indicate a non-conformance in the tested program, but also provide a feedback loop for refining our implementation.

Distilling Executable Parser. We extract the verified parser into OCaml via extraction in Rocq. Our Trusted Computing Base (TCB) includes Rocq (and its extraction process), the OCaml compiler, and the OCaml Bigarray library, which implements our axioms for byte buffers. For practical performance during fuzzing, we directly map Rocq's arbitrary-precision nat type to OCaml's machine integer.

To mitigate overflow risks and ensure the extracted code remains consistent with our formal model, we cap the *basic-bit n* format at 48 bits. While using tools like CertiCoq [11] could reduce the TCB, it can incur more overhead and deviate from our primary focus on parsing logic. We consider our approach sufficiently reliable.

Structure-Aware Fuzzing. We utilize AFL++ [26] as our primary fuzzing engine, while employing LibAFL [27] specifically for baseband testing due to its modularity and seamless integration with our emulation environment. To navigate the complexities of ASN.1, we implement structure-aware fuzzing by incorporating ASNFuzzGen [14], which can linearize ASN.1 messages to an internal byte-level abstraction. This custom mutator starts by linearizing the ASN.1 message, then mutates the resulting byte buffer, and finally delinearizes the byte to reconstruct the final ASN.1 message. By combining this structure-aware approach with traditional havoc mutation, we can explore novel message instances that remain structurally plausible yet purposefully deviate from the specification.

Output Normalization. To properly compare parsers beyond a simple accept/reject check, we must compare their output values. However, it is challenging due to the different internal representations of data (C uses structs; Python uses dictionaries). To address this, we first normalize all outputs to a standardized 3GPP format using the JSON Encoding Rules (JER) [33] as a middle ground. For our own verified parser, we automate this JER serialization using the `ppx_deriving` preprocessor. For the parser-under-test, all have implemented the JSON encoder as part of their toolchain, with the exception of baseband targets.

For these targets where reverse engineering internal data formats is impractical, we instead use a simpler metric: the number of bits each parser consumes. This metric is always available, since any ASN.1 UPER parser must accurately track bit positions to decode values correctly. The extended version [66] further illustrates JER output normalization with the example in Figure 2a. The only caveat is that the reference and target parsers may occasionally reject a message for different internal reasons. But we consider this case out of scope, since both parsers can reject this message.

6 Implementation Details

We estimate developing the 12k lines of code (LoC) Rocq framework and its 1.8k LoC OCaml compiler takes 4-5 person-months, as it requires significant manual proofing despite automation tactics. The varied internal data representations across ASN.1 libraries force us to write custom harnesses for each differential testing target. For closed-source baseband targets, we used Ghidra [39] to identify relevant parsing functions and Unicorn [9] for emulation.

Supported Features. We support ASN.1 structures, including SEQUENCE, CHOICE, and SEQ-OF, all with extensions. Terminal types include BOOLEAN, INTEGER, ENUMERATED, BIT/OCTET STRING, and special strings including UTF8String and IA5String. This also includes features used by the constructs above, including extensions in terminal types, length determinant (Normally Small Length Determinant), and Open Type Fields. We omit encoding rules SET-OF and fragmentation that are barely used in practice. We also omit certain high-level language abstractions, including CLASS, as they do not appear in our target protocols and do not impact the message encoding.

7 Evaluation

We experimentally evaluate our verified parser for its applicability, correctness, and performance with the following key questions:

- Q1. How accurate and precise is the VUPER parser compared to other ASN.1 implementations? (§7.2)
- Q2. How effective is VUPER in identifying non-compliance using our differential testing framework? (§7.3)
- Q3. What are the consequences of the non-compliances? (§7.4)
- Q4. How performant is the verified parser compared with other ASN.1 parsers? (the extended version [66])
- Q5. Baseline Comparison. How does VUPER compare to negative testing and consensus testing?

7.1 Evaluation Setup

Targeted Protocols. To ensure practical relevance, we evaluate our parser on messages from the most impactful use cases of ASN.1 UPER: mobile and vehicular communication. We select the following representative message types for our evaluation:

RRC Messages in 4G/5G. We target complex control-plane messages (DL/UL-DCCH-Message) used for critical signaling, such as handovers, between the User Equipment (UE) and base station [10, 25]. They contain more than 800 submessages and utilize diverse ASN.1 structures.

V2X messages. We test key vehicle-to-everything (V2X) messages from the ETSI ITS standard [23, 24]. Cooperative Awareness Messages (CAMs) periodically broadcast a vehicle’s state (e.g., position, speed), while event-driven Decentralized Environmental Notification Messages (DENMs) warn of specific hazards like accidents. While smaller, the V2X messages introduce distinct elements not found in RRC messages, such as extended INTEGER.

Targeted ASN.1 Systems. Our evaluation includes 7 open-source tools: lionet asn1c [5], pycrate [7], asn1tools [2], TITAN [3], two parsers from srsRAN [1, 4] and rasn [8]; and 4 proprietary ones: Objective Systems’ ASN1C [43] (denoted obj-sys), two versions of ProASN (parsers extracted from Samsung S21 and Google Pixel 6 basebands), and fasn1 [13]. Some are general-purpose ASN.1 parser generators; commercial basebands and srsRAN only work for specific 4G/5G message types; and fasn1 is a black-box message dump. Feature support also varies: asn1tools lacks support for parameterized types used in 5G, so we only test it on V2X messages.

Methodology. We employ a dual-input evaluation strategy to validate both the practical utility and the capability of finding non-compliance using VUPER. First, we test our verified parser against a real-world dataset of 5G RRC traces captured from live UE-base station communications to demonstrate conformance. Second, we evaluate the ASN.1 parsers-under-test using fuzzing with a verified oracle, as discussed in §5. Our fuzzing experiment is done using Intel(R) Xeon(R) Gold 6448H @ 2.4GHz CPUs, and we do 5 runs on one specific message (like 5G DL-DCCH-Message) for 24 hours. Additionally, to minimize false positives arising from specification version discrepancies, we generate all open-source parsers from the same ASN.1 source. While newer releases often utilize extensions for compatibility, they occasionally alter the message structure via mandatory field changes. This remains a concern in proprietary baseband parsers under test since their specific implementation versions are unknown.

7.2 VUPER Accuracy (Q1)

We first evaluate VUPER with two sets of datasets: (1) real-world network operators’ and user devices’ downlink (DL) and uplink (UL) traffic, and (2) traffic generated through fuzzing. And we look for the following aspects for accuracy: overall accept/reject of a message, compatibility, JER encoding, and False Positives.

Real-World Dataset. Our comparison on real-world data yielded two primary observations, as shown in Table 4. First, all baseline parsers accepted every collected message without exception. Second, VUPER accepted the majority of messages but rejected a specific subset. For the messages accepted by both VUPER and the baseline targets, we find no discrepancies in the parsed results. Further investigation into the messages rejected by VUPER reveals that all of them contained explicitly encoded DEFAULT values. Per X.691 [32], ‘encodings of components marked DEFAULT shall always be absent if the value to be encoded is the default value of a simple type’. This indicates that the encoders used by the specific network operator are non-compliant with the standard, a deviation that VUPER correctly identifies while other parsers overlook.

Table 4: Real-world 5G data reject-accept comparison

Message Source	# Msgs	VUPER (Acc-Rej)	Baseline (Acc-Rej)
Country-1 Operator (5G DL)	16644	16644 - 0	16644 - 0
Country-1 Operator (4G DL)	47689	47689 - 0	47689 - 0
Country-2 Operator (5G DL)	18499	14593 - 3906	18499 - 0
Qualcomm baseband (5G UL)	42596	42596 - 0	42596 - 0
Qualcomm baseband (4G UL)	60970	60970 - 0	60970 - 0

Fuzzer Generated Dataset. We summarize the accuracy on the fuzzer-generated traffic for four parsers in Table 7. Over 99% of generated inputs produced no discrepancies, confirming that VUPER is largely conformant with standard implementations. To systematically triage the remaining discrepancies, we apply a diagnostic pipeline that categorizes behavioral differences into three classes: (I) VUPER Rejection V_{rej} and Target Acceptance T_{acc} ; (II) VUPER Acceptance V_{acc} and Target Rejection T_{rej} ; and (III) Dual Acceptance and Semantic Divergence. Class I typically corresponds to injection violations, and Class II/III to surjection violations. However, Class I discrepancy can also result from surjection violations where the target parser accepts a message not because a specific constraint is missing, but because a prior error caused the bit-cursor to shift.

Forward/Backward Compatibility Test. To evaluate the forward/backward compatibility of VUPER, we analyzed its parsing results across different protocol evolutions (5G RRC Release 16 and Release 17) and observed two distinct categories of schema changes: (i) *Standard Extension Changes*: Expected modifications utilizing standard ASN.1 extension markers; and (ii) *Structural Redefinitions*: Alterations to existing message types (e.g., modifying a message definition from an empty SEQUENCE to one containing populated fields), found in 7 message types (the extended version [66]). Redefinitions inherently break backward compatibility and depart from ASN.1 evolution principles, so we filter out such messages.

First, to evaluate VUPER’s forward compatibility and handling of unknown extensions, we generate 28816 Rel-17 messages via ASNFuzzGen [14] and filter them using pycrate [7] to isolate inputs with populated extension fields. We then evaluate this corpus against both Rel-16 and Rel-17 parsers. Table 5 presents the results: most messages are accepted by both implementations, with

Table 5: Forward Compatibility Tests (Inputs with Explicit Extensions)

	Rel-17		
Rel-16 \	Accept-SameVer	Accept-DiffVer	Reject
Accept-SameVer	0	0	0
Accept-DiffVer	25241	0	1623
Reject	0	0	1952

Rows and columns represent Rel-16 and Rel-17 outcomes, respectively. For example, location (Rel-16 Accept-DiffVer, Rel-17 Accept-SameVer) represents the number of messages (25241) accepted by Rel-16 with flag DiffVer and Rel-17 with SameVer.

the Rel-16 parser appropriately marking DiffVer (i.e., different version/release) to denote unknown extensions, thereby confirming forward compatibility. The extended version [66] provides an example message showing the field changes. We note two other expected cases: (1) mutually rejected from fuzzer-generated malformed inputs; (2) rejected from Rel-17 but accepted by Rel-16. In the latter, the malformed data sits within a new extension that the Rel-16 parser correctly bypasses as unknown. But the Rel-17 parser recognizes the field and triggers a decoding failure.

Second, we evaluate backward compatibility by feeding a total of 667825 inputs generated by ASN1FuzzGen of Rel-16 to both parsers (filtered to remove structural redefinitions). Presented in Table 6, this result shows that the Rel-16 parser accepts most inputs as SameVer (i.e., the same version/release), while the Rel-17 parser appropriately flags 21k inputs as DiffVer. The only exception was a specific subset of messages flagged as DiffVer by both implementations. This is because those inputs contain the extension fields undefined by both the Rel-16 and Rel-17 specifications, prompting both parsers to identify a version mismatch.

Table 6: Backward Compatibility Tests (Rel-16 Inputs from Fuzzer)

	Rel-17		
Rel-16 \	Accept-SameVer	Accept-DiffVer	Reject
Accept-SameVer	566701	21175	0
Accept-DiffVer	0	33	0
Reject	0	0	79916

Root Cause Analysis. Our root cause analysis strategy is as follows: First, for Class I errors, we retrofit our verified OCaml parser to *emit semantic error codes*. This allowed us to aggregate thousands of different discrepancies into a set of root causes. Second, for Class II and III discrepancies, we perform manual triage by *dissecting the bit-packets*, supplemented by pycrate [7] to visualize the field offsets. Finally, to confirm a suspected root cause, we conduct *targeted case studies* by injecting a single type of non-compliance, like an out-of-range integer, into a minimal ASN.1 definition or specific protocol messages. This approach reveals how the parser handles individual constraints without the noise of multiple errors. We also conduct source code inspection if possible. **False Positives.** We find a class of false positives stemming from the padding bits within the Open Type field. While an Open Type length is specified in octets, the encapsulated data is bit-encoded. This discrepancy can result in 1–7 redundant bits at the end of the final octet. In the verified parser, to ensure full canonicity, we enforce all the redundant bits to be 0. Because these bits are never decoded into concrete values, the standard does not explicitly

Table 7: Fuzzing statistics for DL-DCCH message

Category	Root Cause	asn1c	pycrate	ProASN	Obj-Sys
Inputs Generated (million)	-	8m	3.3m	1.7m	16m
Saved Discrepancies	-	736	1203	112	1682
I (V_{rej} & T_{acc})	Total	590	1180	85	1674
	W1	7	0	0	0
	S2	38	80	14	96
	S4	0	123	1	207
II (V_{acc} & T_{rej})	Total	71	11	7	0
	R1	71	0	0	0
III (V_{acc} & T_{acc} Diff)	R5	0	0	16	0
False Positive	-	75	12	4	8
False Positive Rate	-	10.2%	1.0%	3.6%	0.5%

require them to be 0. Therefore, we do not consider parsers that omit this check to be inherently non-compliant.

JER Accuracy. There is no guarantee of correctness for the native JER encoders across the tested implementations; a flawed encoder could emit an incorrect JSON representation, artificially inflating the discrepancy rate. We resolved this through root cause analysis to eliminate such false positives. For instance, during our analysis, we discovered and fixed a bug within the srsRAN [4] JER encoder that caused it to emit invalid empty JSON objects. To ensure fairness, we normalize the extension group representation as we compare VUPER with different parser-under-test.

7.3 Identified Issues via VUPER as Oracle (Q2)

By analyzing the discrepancies from testing results, we found 20 types of non-compliance (Table 8), categorized by their respective correctness property violation (§4.3). We begin by discussing some common observations before detailing the specifics.

Non-compliances Among All Existing Parsers. Our evaluations reveal 4 classes of non-compliances (S2, S5, S6, S7) that occur in all parsers tested. Notably, using standard differential fuzzing that relies on the consensus among existing UPER parsers would fail to identify these flaws. This highlights the inherent limitations of pure differential testing and the need for a trusted oracle to properly establish a ground truth.

Non-Compliance related to Extension. A large portion of the non-compliance relates to handling extensions (e.g., R2, R4, S5, S6), as well as Open Type field. These findings empirically confirm that the extension mechanism represents a primary source of implementation complexity. This matches our observation in Challenge C1. **Oracle Limitations.** We acknowledge several limitations of our oracle. The class of false positives has already been discussed in §7.2. We identify three categories of false negatives. The first arises from missing features. If VUPER does not model a particular ASN.1 construct, it cannot flag violations involving that construct. The second category concerns mis-modeled features, where VUPER omits or incorrectly implements a particular check, especially when such checks are missed by both the oracle and all existing parsers. While such cases are possible, we consider them unlikely, as the round-trip property limits the space for silently mis-modeled behavior. Finally, the fuzzer may fail to exercise certain edge cases in the input space. However, since the fuzzer itself is not our main contribution, we do not make claims about its completeness.

Security Impact. We categorize the security impact into three types: (a) *Vulnerabilities*: Exploitable flaws (W1, W2, R4) as we demonstrated in §7.4; (b) *Security Deviations*: Flaws (R2-3, R5-6, S4, S12)

Table 8: Non-compliance with X.691 standard

Flaw	Non-Compliance	X.691	Type	Security Impact	Impacted Implementations
W1	Missing INTEGER Constraint Check	11.5	Weak-Inj	Vul (DoS, MC)	asn1c, asn1tools, rasn
W2	Missing SIZE Constraint Check	11.9.4.1	Weak-Inj	Vul (DoS, MC)	asn1c, asn1tools, rasn
R1	Limited SEQ-OF length	20	Surj-1	NC	asn1c
R2	Incorrectly Rejecting more than 64 exts	19.8	Surj-1	SD	pycrate
R3	Error with UTF8String Constraints	30.6	Surj-2	SD	pycrate
R4	Incorrect Handling of Unknown Ext	19	Surj-2	Vul (DoS)	srsRAN
R5	Incorrect Handling of more than 64 exts	19.8	Surj-2	SD	ProASN
R6	Incorrect Handling of Normally Small Length Det	11.9.3.4	Surj-2	SD	rasn
S1	Overlong Encoding (INTEGER) [†]	11.4.6	Strong-Inj	NC	asn1c, asn1tools, pycrate, rasn, fflas1
S2	Overlong Encoding (Length Determinant)	11.9.3.6	Strong-Inj	NC	asn1c, asn1tools, pycrate, srsRAN, ProASN, TITAN, obj-sys, rasn, fflas1
S3	Overlong Encoding (Normally Small Length Det)	11.9.3.6	Strong-Inj	NC	asn1c, asn1tools, pycrate, srsRAN, ProASN, TITAN, obj-sys, fflas1
S4	Missing Open Type length validation	11.2.2	Strong-Inj	SD	asn1tools, pycrate, srsRAN, ProASN, TITAN, obj-sys, rasn, fflas1
S5	Accepting Extensions all absent	19.9	Strong-Inj	NC	asn1c, asn1tools, pycrate, srsRAN, ProASN, TITAN, obj-sys, rasn, fflas1
S6	Accepting ExtensionAdditionGroup all absent	19	Strong-Inj	NC	asn1c, asn1tools, pycrate, srsRAN, ProASN, TITAN, obj-sys, rasn, fflas1
S7	Accepting Encoding of Default Value	19.5	Strong-Inj	NC	asn1c, asn1tools, pycrate, srsRAN, ProASN, TITAN, obj-sys, rasn, fflas1
S8	Missing SIZE Extension Root Constraint Check [†]	20.4	Strong-Inj	NC	asn1c, pycrate, asn1tools, rasn
S9	Missing INTEGER Extension Root Constraint Check [†]	13.1	Strong-Inj	NC	asn1c, pycrate, asn1tools, rasn
S10	ENUM Extension Root Out-of-Bounds	14.3	Strong-Inj	NC	srsRAN
S11	CHOICE Extension Root Out-of-Bounds	23.8	Strong-Inj	NC	srsRAN
S12	Out-of-Bounds Bit-Cursor for Unknown Extension	19	Strong-Inj	SD	ProASN

X.691: the specification section entry that details this Non-Compliance. **Impact:** the potential effect on the overall system, which depends on the system’s assumptions about the parser. Vul: Vulnerability, NC: Non-Compliance, SD: Security Deviation; DoS: Denial of Service, MC: Memory Corruption,

[†] These results exclude the ProASN and srsRAN parsers, as 5G/4G protocols lack these types, leaving their conformance unknown.

that facilitate auxiliary threats, such as protocol confusion or device fingerprinting; and (c) *Non-Compliance*: Violations (R1, S1–3, S5–11) that undermine canonicity and may risk interoperability failures.

7.3.1 Failure to comply with Weak-Injection. W1: Integer Constraint Check. UPER encodes ranged integers in the minimum number of bits, which can leave more bit-states than range values: INTEGER (1.6) requires 3 bits, but 3 bits represent 8 values. Our testing reveals that while some implementations, such as rasn [8], omit the integer range check entirely, asn1c [5] generally enforces these checks but fails in the case of integer ranges exceeding 32 bits with a positive lower bound (CVE-2025-55398). In this case, asn1c will compile the INTEGER type into an unsigned long, but the decoder fails to check the upper bound.

W2: SIZE Constraint Check. Similarly, some implementations fail to enforce length constraints for SEQUENCE OF, BIT STRING, OCTET STRING, or other restricted character strings.

Impact. Omitting INTEGER range checks can trigger assertion failures when downstream applications validate the value, leading to DoS, or out-of-bounds array indexing, resulting in data corruption. Similarly, SIZE constraint violations can cause critical memory corruption, particularly when implementations rely on these constraints for memory allocation without validating dynamic input lengths. Such vulnerabilities are historically significant; for instance, a 4G RRC decoder in a Samsung baseband [28] accepted invalid lengths, resulting in a stack-buffer overflow.

We find an illustrative legacy example in srsRAN 4G (v18.09). The RRCReconfiguration message contains a drb-ToAddModList field, which is defined to have a maximum of 11 items. The parser processes this by decoding a 4-bit length field into n and iteratively populating a pre-allocated array of size 11. However, the implementation fails to validate whether n falls in this range, allowing an attacker to supply a valid 4-bit integer exceeding 11 and trigger an immediate out-of-bounds memory write.

7.3.2 Failure to comply with Surjection. We identify two types of Surjection rule violations. First, the decoder may fail to decode a

valid message. For example, asn1c caps SEQUENCE OF types at 200 elements. Code inspection reveals this limit protects against SEQUENCE OF NULL compression bombs. This could cause interoperability issues, as valid messages might be rejected. Second, decoders may misinterpret inputs, i.e., accepting a message but mapping it to an incorrect internal representation.

R4: Incorrect Handling of Unknown Extension. The srsRAN decoder fails to process unknown extensions correctly when an ASN.1 definition includes an extension marker (...) but no defined extension additions. When encountering a message from a future protocol version that contains extension data, the srsRAN decoder ignores the extension bit entirely and attempts to parse the subsequent field at the bit position where the extension should have been. This causes the decoder to misinterpret the remaining bitstream.

R5: Incorrect Handling of Extension Count. ProASN exhibits a similar problem in its handling of extension counts. According to the standard, the number of extensions is encoded as a normally small length determinant: a leading bit 0 indicates a count of up to 64, encoded in the subsequent 6 bits, while a leading 1 implies that an explicit length determinant follows. The issue arises that ProASN ignores the prefix bit, essentially always interpreting the next 6 bits as the count. Thus, if the sender includes more than 64 items in a message, ProASN would obtain an incorrect offset and misinterpret the remainder of the message.

Impact. Such deviations can cause interoperability failures between independently developed systems. A single field error can corrupt all subsequent decoding, creating logical inconsistencies for security-critical parameters. These failures can also lead to DoS if systems fail to establish or maintain a valid connection.

7.3.3 Failure to comply with Strong-Injection. These findings detail violations of the *Strong-Injection* property caused by non-compliant ASN.1 implementations or lenient interpretations of the X.691 specification. Some violations could present interoperability risks and potential security vulnerabilities.

Handling Difference. One issue is that vendors may handle messages inconsistently, with one system accepting a message that another rejects, resulting in potential unexpected behaviors. For example, the length determinant in Open Type field indicates how many bytes the encoding takes. However, when the length determinant mismatches the actual payload, for instance, a 3-byte declaration for 2 bytes of data, the parsers might differ in how they handle it. Pycrate [7] would skip only 2 bytes in total and immediately start parsing the next field, while ProASN would skip 3 bytes in total and then start parsing the next field.

Alternative Encodings. Lenient parsers may accept multiple distinct encodings of the same value, violating non-malleability. For instance, unconstrained INTEGERS should not have the first 9 bits to be all zeros or all ones, as this indicates redundant leading octets. Nonetheless, we find that many parsers accept positive values padded with extraneous leading 0×00 bytes.

Non-Compliant Encoder. VUPER also reveals non-conformant encoders. Beyond the encoding of the DEFAULT field by one of the network operators (§7.2), we find similar issues in the srsRAN [4] and rasn [8]. Per X.691 standard, an 'ExtensionAdditionGroup' must be encoded as a missing extension addition if all its components are missing. However, these encoders fail to omit the group when it is marked present but contains only absent items.

Impact. First, *strong-injection* violations could cause interoperability issues. When implementations exhibit differential handling of alternative encodings, a single broadcast message processed differently by receiving devices can lead to state desynchronization, mirroring the impact of surjection violations. Similarly, a non-compliant encoder may emit messages that strictly conforming parsers reject, causing interoperability issues. For instance, during 5G handover, if a source gNB incorrectly encodes a relayed Reconfiguration message from the target gNB, a UE employing a strict parser will discard the packet, resulting in handover failure.

Second, these violations facilitate device fingerprinting. An active attacker with a fake base station can send specific alternative encodings to a device and observe which are accepted/rejected. The resulting accept/reject signature can often reveal the device manufacturer. For example, Flaw S12 is unique to ProASN in our findings. An attacker can craft an RRC message containing an unknown extension whose declared length exceeds the actual payload. If the device accepts and responds, the attacker can confirm that the target uses a Samsung baseband. Such fingerprinting further enables the attacker to exploit device-model specific known vulnerabilities, e.g., authentication bypass in Samsung Galaxy S21 [57].

7.4 Proof-of-Concept Attacks (Q3)

We present four proof-of-concept attacks demonstrating how low-level parser errors propagate to the downstream protocol layer and induce real-world security implications.

7.4.1 Injection Violation Exploits. We identified two vulnerabilities in the OpenAirInterface [6] implementation of the 4G and 5G RRC layers by exploiting Flaw W1 and W2 of the asn1c.

Out-of-Bound PLMN IdentityList causing UE crash. The first issue, shown in Figure 5a, involves the PLMN (Public Land Mobile Network) Identity List. This is a critical data field within the SIB1 (System Information Block Type 1) broadcast from a cell tower to

announce the carrier networks it supports. The 4G RRC specification limits this list to a maximum of 6. However, a malicious eNB can broadcast a SIB1 message containing an oversized list with 7 or more identities, which the OAI UE stack fails to reject during initial parsing. The UE then needs to store the index of its chosen carrier to maintain state across the asynchronous connection procedure. This stored index is later used when the UE needs to inform the network which PLMN it has selected to complete the handshake. But the UE would fail to encode the stored index into the selectedPLMN-Identity field, defined as INTEGER (1..6). This results in an assertion failure, allowing a malicious eNB to remotely crash the UE's RRC protocol stack.

Constraint violation message causing gNB crash. The second vulnerability (Figure 5b) can be exploited during a 5G handover [10] from the source gNB_A to the target gNB_B. In this procedure, gNB_A initiates a handover based on the UE Measurement Report and awaits a HandoverCommand from gNB_B. This command contains an RRCReconfiguration message intended for the UE to complete the handover. A compromised target node can inject a malformed RRCReconfiguration message that contains either a SIZE or INTEGER constraint violation like an overlong DRB-ToAddModList. Instead of rejecting the malformed packet during the initial decoding phase, gNB_A accepts the packet and immediately attempts to re-encode it as a DL-DCCH-Message for over-the-air transmission. This re-encoding process fails as the encoder performs its own strict constraint checks; however, rather than gracefully handling the error, the system triggers a fatal assertion failure that crashes the program, resulting in a DoS on the source base station. A compromised femtocell [63] can cause such DoS by acting as the target gNodeB as shown in Figure 5b.

7.4.2 Surjection Violation Exploits. Such surjection violations can lead to subtle yet critical miscommunications; although they may not trigger immediate system crashes, they can introduce severe logical errors. We discuss two such instances below.

Miscommunication between UE and gNB. The first issue is a miscommunication during an RRCReconfiguration procedure between the UE and gNB within the srsRAN 5G implementation. Specifically, the RadioBearerConfig Information Element configures the keys for Data Radio Bearers (DRBs) using critical security fields, including keyToUse (selecting between the Master Key K_{gNB} or Secondary Key $S - K_{gNB}$) and sk-Counter (providing freshness for secondary key derivation). We observe that certain message extensions could cause the UE to misinterpret critical security parameters, for example, setting keyToUse from the secondary key to the master key. This creates a mismatched security context and would cause the integrity and deciphering checks in DRBs to fail, effectively resulting in a session-terminating DoS.

Miscommunication during eNB handover. The second issue occurs during a 4G handover, as illustrated in Figure 5c. Similar to the 5G handover, the source eNB would signal the transfer by forwarding a RRCReconfiguration message from the target eNB to the UE. This message contains the specification of the new physical cell and critical SecurityConfig parameters, such as the selected encryption and integrity protection algorithms. We find that encodings involving unknown extensions can drive the UE into an inconsistent configuration state. As a result, the UE fails to synchronize

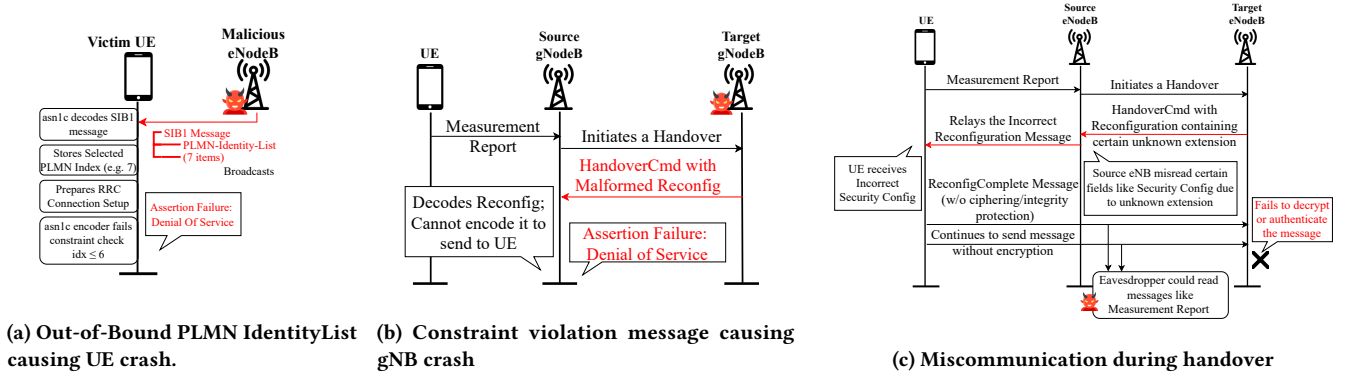


Figure 5: Proof-of-Concept Attacks

with the target eNB due to invalid physical-layer parameters, causing handover failure. Moreover, inconsistencies in the interpreted security configuration can lead to a ciphering/integrity mismatch, prompting the target eNB to discard subsequent UE signaling. This bug may also induce information leakage if the UE mistakenly adopts a lower security level than intended by the network.

7.5 Baseline Comparison

To evaluate the unique contribution of VUPER as a verified test oracle, we compare its bug-detection capabilities against two baselines. The first is *negative testing*: deliberately generating invalid inputs for the parser to check whether it rejects them. The second is *consensus testing*, i.e., differential testing without a verified oracle, where a bug is flagged when implementations disagree.

To enable a controlled comparison, we focus the baseline evaluation on 5 open-source implementations: `asn1c`, `pycrate`, `rasn`, `asn1tools`, and `TITAN`. Open-source parsers are well-suited for two reasons: they can be compiled against a single ASN.1 specification (whereas baseband and `srsRAN` parsers are tied to specific protocol versions), and source access makes it easier to triage. Because `asn1tools` and `rasn` do not support 5G formats, we evaluate on V2X protocols. This restricts the comparison to 14 of the 20 bug types found by VUPER (W1-2, R2-3, R6, S1-9), including the four bugs (S2, S5-7) present in all tested parsers.

Negative Testing. We use OTABase [45], an existing negative test case generator for the RRC layer. However, OTABase is originally designed to test the entire 4G RRC layer (i.e., both the parser and packet-processing logic) rather than strictly evaluating parsers. Hence, we adapt it and introduce the following changes.

First, we filter the mutation strategies to exclusively generate malformed inputs. We discard strategies that produce syntactically valid ASN.1 messages, like generating empty sequences that can test system boundaries, but fail to violate any parser constraints. Concretely, this yields the following negative test case generation strategies: (1) Out-of-range values for INT/String; (2) Out-of-set indices for ENUM/CHOICE; (3) Length-content mismatches in SEQ-OF; (4) Random bit truncation from valid packets. Second, we extend OTABase to support V2X message generation.

Our evaluation generates over 10k test cases that cover all 4 types of generation strategies. Our testing reveals that OTABase can

Table 9: Bug Finding Baseline Comparison on V2X Protocols

Methodology	Logic Coverage	Found Bugs (V2X Subset)
Negative Testing	Primitive Bounds	W1, W2
Consensus Testing	Implementation Differences	W1, W2, R6, S8, S9
VUPER as Oracle	Roundtrip-Property	All 14 Types

identify only W1 and W2, but misses the remaining vulnerabilities. This limitation arises primarily because OTABase focuses on testing RRC layer and mutating primitive data types and lacks the specific, schema-aware mutations necessary to trigger deep parsing bugs. For instance, ASN.1 extensions remain unexplored.

Consensus Testing. We conducted a differential fuzzing campaign against the 5 aforementioned open-source parsers. By cross-referencing the accept/reject status and the decoded values for every input, we utilize the resulting discrepancies to identify non-compliance bugs. We, however, notice that a limitation of consensus testing is the triaging overhead, as we cannot use the semantic error codes from VUPER. We solve this challenge by tracking the semantic error code emitted by some parsers. However, they often force us to revert to the labor-intensive manual triaging.

Ultimately, this campaign identified five distinct bugs: W1, W2, R6, S8, and S9. This result is due to several reasons. First, differential testing inherently relies on implementation divergence, so non-compliances shared by all parsers are invisible. Second, this method suffers from parser-choice bias. It cannot expose bugs unique to untested parsers. Finally, this approach is subject to outlier shadowing. When a particular parser like `rasn` introduces a disproportionate number of behavioral differences, it adds noise to the triaging process and could mask other issues.

Summarized in Table 9, nine bugs, including the four unique bugs (S2, S5, S6, and S7), evade both negative and consensus testing, making them uniquely discoverable by VUPER.

8 Discussion

Incorrect Usage of ASN.1 libraries. Our analysis also uncovered crashes in the RRC implementation of OAI [6] resulting from unsafe assumptions in the application code. These issues are not inherent to the parser but are related to mixing protocol-level assumptions with the parser’s ability to sanitize inputs. One (CVE-2025-55396) is an out-of-bounds access due to a mismatch in the size of a SEQ-OF

between the 5G NR specification and the OAI's implementation. The other (CVE-2025-55397) involves a null pointer dereference where OAI assumes an optional RRC protocol field is always present.

Bugs in ASN.1 Compiler. While our primary objective is not to audit ASN.1 compilers, our evaluation of diverse ASN.1 definitions revealed 2 compiler-level bugs. One bug is in `rasn` [8] where the compiler erroneously generates import statements for parameterized types that have already been unfolded. The second bug is in `pycrate` [7] where the compiler crashes when a `SEQUENCE-OF` type is defined with multiple constraints stacked together.

Applicability to other Encoding Rules. Our methodology extends to other encoding rules such as `APER`, though adaptation poses challenges. Because `APER` is not consistently byte-aligned, its encoding varies based on the current bit position, making the *encode-invariance* property difficult to satisfy. Therefore, it is practical to focus exclusively on the weak-injection property. Despite these differences, our findings directly transfer to `APER`: the `asn1c` [5] `APER` decoder also lacks `INTEGER` constraint checks.

Responsible disclosure. We reported our findings to the respective library maintainers, receiving varied responses and 3 assigned CVEs, with several pending. At least 3 bugs have already been patched, including `asn1c`'s flaw W1, W2 and `pycrate`'s R3, with more detailed report in the extended version [66]. A notable discussion emerged over non-canonical compliance, where `asn1c` developers argue for the robustness principle [16]; other developers, e.g., Objective System, welcome our detailed edge-case examples, viewing them as valuable test cases for their implementation of the Canonical UPER decoder. This reinforces VUPER's potential as a reference implementation for improving parser security.

9 Related Work

Verified Parser. Verified parsers are an active area of research that bridges formal verification, programming languages, and security. `EverParse` [47, 54] and `Compars` [62] are non-malleable parser libraries built in `F*` [53]. `EverParse` generates efficient, zero-copy C code, while `Compars` focuses on integration with cryptographic frameworks like `DY*` [15]. `Roq` [56] is used in both general frameworks like `Narcissus` [22] and specific applications like the Protocol Buffer compiler [64]. Other notable works include `Vest` [18], a verified Rust parser using `Verus` [35]; `PulseParse` [48], a non-malleable parser for CBOR formats; and `Bigrammar` [55], an instruction decoder that satisfies *surjection* and *weak-injection*.

ASN.1 Parsers. The widespread use of ASN.1 has prompted many formalization efforts, primarily targeting the Distinguished Encoding Rules (DER), a byte-aligned encoding fundamentally different from UPER. `ASN1*` [42] uses `EverParse` [47] to generate non-malleable DER parsers. Formalization of X.509 [30] has also yielded verified DER parsers as a byproduct [21, 36]. Beyond DER, existing efforts include a UPER parser for vehicle communications [59], but it lacks essential features, and a verified parser for the non-standard, legacy ASN.1/ACN scheme [17] with no injection property. A comprehensive comparison of these projects is in Table 1.

Differential Testing & Fuzzing. Differential testing [37] has been successfully applied for cryptographic libraries [61], X.509 validation [20, 21, 36, 46] and ZIP parser [65]. Our approach of using a trusted reference is similar to that of `ARMOR` [21] or `Verdict` [36].

10 Conclusion

In this paper, we present VUPER, a verified parser/serializer framework for ASN.1 UPER, one of the widely used interface description languages for complex networks. VUPER provides bit-level reasoning and round-trip properties to ensure parser correctness and security, support for forward/backward compatibility, and verified ASN.1 combinators. We develop a testing framework using VUPER as a verified oracle. Our experiments uncovered 20 types of inconsistencies in 7 open-source and 4 commercial implementations. This work highlights the need for more robust, high-assurance parsers.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their feedback and suggestions. We also thank the corresponding developers for cooperating with us during our responsible disclosure. This paper was edited for grammar using Claude and Gemini. This work has been supported by the NSF under grants 2145631, 2215017, and 2614551, and the Public Wireless Supply Chain Innovation Fund (PWSCIF) under Federal Award ID Number 51-60-IF007.

References

- [1] 2020. *srsRAN-4G*. https://github.com/srsran/srsRAN_4G
- [2] 2024. *asn1tools*. <https://github.com/eerimoq/asn1tools>
- [3] 2024. Eclipse TITAN (11.0.0). <https://gitlab.eclipse.org/eclipse/titan/titan.core>
- [4] 2024. *srsRAN Project (24.10)*. https://github.com/srsran/srsRAN_Project
- [5] 2025. *lionet asn1c (mouse07410 fork)*. <https://github.com/mouse07410/asn1c>
- [6] 2025. *OpenAirInterface*. <https://gitlab.eurecom.fr/oai/openairinterface5g>
- [7] 2025. *Pycrate*. <https://github.com/pycrate-org/pycrate>
- [8] 2025. *rasn*. <https://github.com/librasn/rasn>
- [9] 2025. *Unicorn Engine*. <https://github.com/unicorn-engine/unicorn>
- [10] 3GPP. 2024. NR; Radio Resource Control (RRC) Protocol Specification.
- [11] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. *CertiCoq: A verified compiler for Coq*. In *The 3rd workshop on Coq for programming languages*.
- [12] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785
- [13] Fabrice Bellard. 2023. *ffasn1*. <https://bellard.org/ffasn1/>
- [14] Nathaniel Bennett, Weidong Zhu, Benjamin Simon, Ryon Kennedy, William Enck, Patrick Traynor, and Kevin R. B. Butler. 2024. RANsacked: A Domain-Informed Approach for Fuzzing LTE and 5G RAN-Core Interfaces. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. 2027–2041.
- [15] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseini, Ralf Küsters, Guido Schmitz, and Tim Würtele. 2021. *DY**: A Modular Symbolic Verification Framework for Executable Cryptographic Protocol Code. In *2021 IEEE European Symposium on Security and Privacy*. 523–542.
- [16] Robert Braden. 1989. Requirements for Internet Hosts - Communication Layers. Internet Requests for Comments. <https://www.rfc-editor.org/info/std3>
- [17] Mario Bucev, Samuel Chassot, Simon Felix, Filip Schramka, and Viktor Kunčák. 2025. Formally Verifiable Generated ASN.1/ACN Encoders and Decoders: A Case Study. In *Verification, Model Checking, and Abstract Interpretation*. 185–207.
- [18] Yi Cai, Pratap Singh, Zhengyao Lin, Jay Bosamiya, Joshua Ganchar, Milijana Surbatovich, and Bryan Parno. 2025. *Vest*: Verified, secure, high-performance parsing and serialization for Rust. In *34th USENIX Security Symposium*.
- [19] CVE. 2025. Common Vulnerabilities and Exposures (CVE) - ASN.1 Search Results. <https://www.cve.org/CVERecord/SearchResults?query=asn.1>
- [20] Joyanta Debnath, Sze Yiu Chau, and Omar Chowdhury. 2021. On Re-engineering the X.509 PKI with Executable Specification for Better Implementation Guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1388–1404. doi:10.1145/3460120.3484793
- [21] 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 *IEEE Symposium on Security and Privacy*. 1462–1480.
- [22] 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, Article 82 (July 2019), 29 pages. doi:10.1145/3341686

- [23] ETSI. 2014. Intelligent Transport Systems; Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service.
- [24] ETSI. 2014. Intelligent Transport Systems; Vehicular Communications; Basic Set of Applications; Part 3: Specifications of Decentralized Environmental Notification Basic Service.
- [25] ETSI. 2018. LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification.
- [26] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies*. USENIX Association.
- [27] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM Conference on Computer and Communications Security*. 1051–1065.
- [28] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. 2022. FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *Symposium on Network and Distributed System Security*.
- [29] Syed Rafiul Hussain, Mitziu Echeverria, Intiaz Karim, Omar Chowdhury, and Elisa Bertino. 2019. 5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 669–684.
- [30] ITU. 2019. X.509 information technology—open systems interconnection—the directory: Public-key and attribute certificate frameworks.
- [31] ITU. 2021. Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.
- [32] ITU. 2021. X.691 : Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER) .
- [33] ITU. 2021. X.697 : Information technology - ASN.1 encoding rules: Specification of JavaScript Object Notation Encoding Rules (JER).
- [34] Hongil Kim, Jiho Lee, Eunhyu Lee, and Yongdae Kim. 2019. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *2019 IEEE Symposium on Security and Privacy*. 1153–1168. doi:10.1109/SP.2019.00038
- [35] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 85 (April 2023), 30 pages. doi:10.1145/3586037
- [36] Zhengyao Lin, Michael McLoughlin, Pratap Singh, Rory Brennan-Jones, Paul Hitchcox, Joshua Ganther, and Bryan Parno. 2025. Towards practical, end-to-end formally verified x.509 certificate validators with Verdict. In *34th USENIX Security Symposium*.
- [37] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.
- [38] National Institute of Standards and Technology. 2024. CVE-2024-5535.
- [39] National Security Agency. 2025. Ghidra Software Reverse Engineering Framework (11.4). <https://ghidra-sre.org/>
- [40] National Vulnerability Database. 2009. CVE-2009-2408.
- [41] National Vulnerability Database. 2019. CVE-2019-6740.
- [42] 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 Conference on Certified Programs and Proofs*. 275–289. doi:10.1145/3573105.3575684
- [43] Objective Systems, Inc. 2025. ASN1C ASN.1 Compiler (v7.9). <https://obj-sys.com/products/asn1c/>.
- [44] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunhyu Lee, Insu Yun, and Yongdae Kim. 2022. {DoLTest}: In-depth downlink negative testing framework for {LTE} devices. In *31st USENIX Security Symposium*. 1325–1342.
- [45] CheolJun Park, Marc Egli, BeomSeok Oh, Tuan Dinh Hoang, Suhwan Jeong, Martin Crettol, Insu Yun, Mathias Payer, and Yongdae Kim. 2025. OTABase: Enhancing Over-the-Air Testing to Detect Memory Crashes in Cellular Basebands. In *2025 IEEE Annual Computer Security Applications Conference*. IEEE, 291–307.
- [46] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy*. 615–632. doi:10.1109/SP.2017.27
- [47] 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*. 1465–1482.
- [48] Tahina Ramananandro, Gabriel Ebner, Guido Martínez, and Nikhil Swamy. 2025. Secure Parsing and Serializing with Separation Logic Applied to CBOR, CDDL, and COSE. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 2744–2758.
- [49] SAE International. 2020. V2X Communications Message Set Dictionary (SAE Standard J2735_202007). https://www.sae.org/standards/content/j2735_202007/.
- [50] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. 2019. New vulnerabilities in 4G and 5G cellular access network protocols: exposing device capabilities. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. 221–231.
- [51] Simon Sharwood. 2025. Korean telco failed at femtocell security, exposed customers to snooping and fraud. https://www.theregister.com/2025/12/30/kt_telecom_femtocell_security_fail/
- [52] Paul Steckler. 2007. A Formal Semantics for ASN.1. In *2007 High Confidence Software and Systems Conference*. <https://sos-vo.org/node/2313>
- [53] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages*. 256–270.
- [54] 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*. 31–45.
- [55] Gang Tan and Greg Morrisett. 2018. Bidirectional Grammars for Machine-Code Decoding and Encoding. *J. Autom. Reason.* 60, 3 (March 2018), 257–277.
- [56] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.20.0. <https://coq.inria.fr/doc/V8.20.0/refman>.
- [57] Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain. 2024. Logic gone astray: A security analysis framework for the control plane protocols of 5G basebands. In *33rd USENIX Security Symposium*. 3063–3080.
- [58] Mark Tullsen. 2019. ASN. 1 Encoding Schemes Done Right Using CMPCT.. In *Bx@ PLW*. 1–15.
- [59] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 2018. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Computer Aided Verification*. 413–429.
- [60] Marcell van Geest and Wouter Swierstra. 2017. Generic packet descriptions: verified parsing and pretty printing of low-level data. In *Proc. of the 2nd ACM Workshop on Type-Driven Development*. 30–40. doi:10.1145/3122975.3122979
- [61] Guido Vranken. 2025. Cryptofuzz. <https://github.com/MozillaSecurity/cryptofuzz>
- [62] Théophile Wallez, Jonathan Protzenko, and Karthikeyan Bhargavan. 2023. Compare: Provably Secure Formats for Cryptographic Protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 564–578. doi:10.1145/3576915.3623201
- [63] Yaru Yang, Yiming Zhang, Tao Wan, Haixin Duan, Deliang Chang, Yishen Li, and Shujun Tang. 2026. Small Cell, Big Risk: A Security Assessment of 4G LTE Femtocells in the Wild. In *Network and Distributed System Security Symposium*.
- [64] Qianchuan Ye and Benjamin Delaware. 2019. A verified protocol buffer compiler. In *Proceedings of the 8th ACM Conference on Certified Programs and Proofs*. 222–233.
- [65] Yufan You, Jianjun Chen, Qi Wang, and Haixin Duan. 2025. My ZIP isn't your ZIP: identifying and exploiting semantic gaps between ZIP parsers. In *34th USENIX Security Symposium*. Article 23, 20 pages.
- [66] Xiaotian Zhou, Kai Tu, Ali Ranjbar, Yilu Dong, Gang Tan, and Syed Rafiul Hussain. 2026. VUPER: Verified ASN.1 UPER Parser (Extended Version). <https://github.com/SyNSec-den/VUPER>

Ethical Considerations

This work develops a formally verified ASN.1 UPER parser and uses it to uncover non-conformance in major implementations. The principal ethical risk is that disclosing such flaws could aid attackers. We mitigated by reporting every finding to the respective maintainers with proof-of-concept inputs and offering assistance with remediation. The real-world impact (e.g., DoS, authentication bypass) depends on how each parser is integrated downstream, and we provide no weaponized exploits, limiting risk to end users. All experiments were legal and run in isolated, self-hosted testbeds, affecting no external systems or live networks. We assess that the security benefits to parser developers, dependent vendors, end users, and the research community outweigh these mitigated risks.

Open Science

To support open science, we release our artifacts publicly [66]. The release includes VUPER's verified parser/serializer, the parser compiler and the differential-testing harnesses.