ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Formally Verified ASN.1 Python Encoders and Decoders

Master Thesis

Luca Schafroth

March 23, 2026

Advisors: Prof. Dr. Peter Müller, Dr. Marco Eilers, Simon Felix
Department of Computer Science, ETH Zürich

Abstract

In safety-critical domains such as aerospace and satellite systems, testing alone cannot guarantee the correctness of communication protocol implementations. ASN.1 and its companion language ACN are widely used standards for defining platform-independent data structures, and the ESA-developed compiler ASN1SCC generates encoders and decoders from these specifications for several target languages. Python's growing adoption in the space industry has motivated the development of a Python backend for ASN1SCC by Ateleris GmbH.

This thesis applies formal verification to the generated Python artifacts using Nagini, a static verifier for Python based on the Viper infrastructure. The central contribution is the full verification of the `BitStream` data structure, the core runtime component shared by all generated codecs, establishing absence of runtime errors and functional correctness of all read and write operations. Building on this, we prove round-trip correctness — that encoding followed by decoding recovers the original value — for the types `BOOLEAN`, `ENUMERATED`, `NULL`, constrained `INTEGER`, fixed-size `OCTET STRING`, and `SEQUENCE` with fixed-size, non-optional fields under the unaligned Packed Encoding Rules (uPER).

The key architectural contribution enabling these proofs is a *segment abstraction* that decouples bit-level buffer reasoning from codec-level correctness arguments, making individual proof obligations tractable and supporting a compositional round-trip argument at the top level. We also extended the ASN1SCC Python backend to automatically emit the required Nagini annotations for generated data classes. Finally, we contributed several extensions to Nagini itself — support for `bytearray`, shift operators, dataclasses, and `IntEnum` — broadening the language features available for verifying Python programs beyond this project.

These results demonstrate that the class of correctness guarantee established by prior work for the Scala backend of ASN1SCC is attainable for Python, providing a mathematically rigorous foundation for safety-critical applications.

Acknowledgments

First and foremost, I would like to thank my advisor, Marco Eilers, for his constant guidance and support throughout this thesis. Without his intricate understanding of Nagini, Viper, and formal verification as a whole, this project would not have been possible. I am especially grateful for his patience, encouragement, and determination in overcoming some of the seemingly hopeless hurdles we encountered along the way. He was the best advisor I could have hoped for.

I would also like to thank Prof. Dr. Peter Müller for his openness to take on this project and for the opportunity to work in his group.

I thank Simon Felix for introducing me to the topic of this project and for facilitating the collaboration between ETH Zürich, Ateleris GmbH, and the European Space Agency.

I am grateful to my colleagues Julia Hartmann and Manuel Stutz at Ateleris for their support in integrating the verification changes into the ASN1SCC project.

I am also grateful to Xenia Füchter, Fabio Rentsch, and Rolf Schafroth for carefully proofreading this thesis and helping to correct mistakes in the written text.

Finally, I would like to thank my family for their encouragement and support throughout my studies, and in particular Xenia.

Contents

Chapter 1

Introduction

In safety-critical domains such as aerospace and satellite systems, software defects in communication protocols can have catastrophic consequences. Testing alone cannot prevent such defects: no finite test suite can guarantee correctness for all possible inputs and operational conditions. Formal verification offers a stronger alternative by proving correctness mathematically. This need becomes even more pronounced when software is generated automatically from specifications: the sheer volume of generated code makes manual review impractical, and a scalable verification approach is needed to establish trust in the outputs.

The Abstract Syntax Notation One (ASN.1) [14] and its companion language ASN.1 Control Notation (ACN) [18] exemplify this setting. ACN allows customization of the bit encoding of data structures, while defaulting to the unaligned Packed Encoding Rules (uPER) [15] when no ACN specifications are provided. Together, these standards provide a framework for defining data structures exchanged across diverse platforms and programming languages, and are relied on extensively by the European Space Agency (ESA) in its missions. ASN1SCC [12] is an ESA-developed open-source compiler that translates ASN.1/ACN specifications into encoders and decoders for C, Ada, and Scala, specifically designed for embedded and space applications. Bucev et al. [5] formally verified the Scala backend using Stainless [9, 10], proving that the generated encoders and decoders correctly implement their specifications.

Python's growing adoption in the space industry for ground systems, data analysis, and simulation has created demand for a Python backend in ASN1SCC. Ateleris GmbH is currently developing this backend. This thesis extends that effort by applying formal verification to the generated Python artifacts using Nagini [7, 8], a static verifier for Python built on the Viper [13] infrastructure, with the goal of establishing correctness guarantees similar to those Bucev et al. achieved for Scala.

Contributions

This thesis makes the following contributions:

1. **Verified BitStream implementation**
   We fully verified the `BitStream` data structure, a core component of the Python runtime library that is shared by all generated encoders and decoders. The verification establishes basic safety properties such as absence of runtime errors (e.g., index out-of-bounds accesses), as well as full functional correctness of all read and write operations: each written value is correctly retrieved by a subsequent read, and previously written data remains unchanged. To make the verification tractable, we designed a layered abstraction architecture that avoids bit-level reasoning at higher layers.
2. **Verified uPER codecs for a representative set of types**
   We proved the absence of runtime errors and round-trip correctness (i.e., that encoding followed by decoding recovers the original value) for the types `BOOLEAN`, `ENUMERATED`, `NULL`, constrained `INTEGER`, fixed-size `OCTET STRING`, and `SEQUENCE` with fixed-size, non-optional fields. The proof structure is designed to extend naturally to the remaining ASN.1 types.
3. **Extensions to Nagini**
   We extended Nagini with several features required to verify the Python code, including support for `bytearray`, left and right shift operators, dataclasses, and `IntEnum`.

Chapter Overview

Chapter 2 introduces the relevant background on ASN.1/ACN, the Nagini–Viper verification stack, and prior verification work on ASN1SCC. Chapter 3 presents the verified Python runtime library, including the `BitStream` implementation and the `Encoder` and `Decoder` built on top of `BitStream`, covering the layered abstraction from raw bit operations up to the uPER codec helpers. Chapter 4 describes the extensions to the ASN1SCC backend and the verification of the generated Python encoders and decoders. Chapter 5 details the extensions made to Nagini during this project. Chapter 6 discusses proof-structuring techniques and challenges encountered during the verification work. Chapter 7 evaluates the verification effort and compares it with the work of Bucev et al. [5] on the Scala counterpart. Finally, Chapter 8 summarizes the contributions of this thesis and outlines directions for future work.

Chapter 2

Background

This chapter provides background on the relevant topics and technologies for this thesis. Familiarity with the Python programming language, formal verification, and SMT solvers is assumed. We first describe the Abstract Syntax Notation One (ASN.1) and the European Space Agency's ASN.1 compiler, ASN1SCC. We then introduce the relevant formal verification tools, namely Viper, Nagini, and Stainless.

2.1 Abstract Syntax Notation One

The Abstract Syntax Notation One (ASN.1) is an interface description language for defining protocol messages that can be serialized and deserialized in a cross-platform manner. Originally standardized in 1984 by the International Telecommunication Union (ITU-T X.680 [14]), ASN.1 is widely used in telecommunications and computer networking protocols. In particular, it is used by the European Space Agency (ESA) in the formal specification of its PUS-C standard [1, 17].

2.1.1 Overview and Motivation

ASN.1 addresses a fundamental challenge in distributed systems: how to exchange structured data between heterogeneous systems that may use different hardware architectures, operating systems, and programming languages. It provides a formal and unambiguous notation for describing data structures independently of any particular encoding or programming language. Furthermore, it separates the abstract structure of data from its concrete representation, delegating the latter to encoding rules. This separation allows the same abstract data structure to be encoded in different ways depending on the requirements of a specific use case (e.g., optimizing for compactness, readability, or ease of processing).

2.1.2 ASN.1 Data Types

ASN.1 provides a range of primitive and structured types. Primitive types include `BOOLEAN`, `ENUMERATED`, `INTEGER`, `NULL`, and `REAL`, as well as string variants such as `BIT STRING`, `IA5String`, and `OCTET STRING`. Structured types allow primitive values to be composed: `SEQUENCE` groups a fixed set of named fields analogously to a record, `SEQUENCE OF` represents a homogeneous array, and `CHOICE` encodes a tagged union. The complete ASN.1 type system [14] encompasses additional constructs beyond those listed above.

Listing 1 illustrates how these types compose into a larger definition. The `Message` type is a `SEQUENCE` whose `priority` field references a separately defined `ENUMERATED` type. The `messageId` field is constrained to values in the range $[0, 65535]$, while the `timestamp` field has no such constraints. Finally, `payload` can hold any arbitrary number of octets.

```
Message ::= SEQUENCE {
    messageId   INTEGER (0..65535),
    timestamp   INTEGER,
    payload     OCTET STRING,
    priority    Priority
}

Priority ::= ENUMERATED {
    low(0),
    normal(1),
    high(2),
    critical(3)
}
```

Listing 1: ASN.1 specification composing a `SEQUENCE` with a referenced `ENUMERATED` type.

2.1.3 Unaligned Packed Encoding Rules (uPER)

ASN.1 itself defines only the abstract syntax, while several encoding schemes — such as ACN, BER, DER, or PER — determine the binary representation. The Packed Encoding Rules (ITU-T X.691 [15]) are designed to produce compact encodings suitable for bandwidth-constrained environments such as wireless communications and space missions. In particular, the unaligned variant, uPER, achieves minimal size through several techniques:

1. **No Padding**
   uPER does not insert padding bits to align fields to byte boundaries, allowing maximum density at the cost of more complex encoding and decoding logic.

2. **Constraint Exploitation**
   When a type has constraints, uPER uses only the minimum number of bits needed to represent values within that range. For a constrained integer with range $[min, max]$, uPER uses $\lceil \log_2(max - min + 1) \rceil$ bits.
3. **Omitting Tags**
   Type information is not encoded when it can be inferred from the schema. For example, in a `SEQUENCE`, field types are known from the schema, so no type tags are needed.
4. **Length Determinant Optimization**
   When the length of a variable-size type can be determined from constraints, uPER encodes it compactly. For example, a constrained `OCTET STRING` with a known maximum length uses the minimal number of bits necessary to encode the actual length.

Example

Consider encoding a `BOOLEAN` and an `INTEGER` constrained to the range $[0, 7]$ as shown in Listing 2.

```
Data ::= SEQUENCE {
    flag        BOOLEAN,
    value       INTEGER (0..7)
}
```

Listing 2: ASN.1 specification with `BOOLEAN` and `INTEGER`.

In uPER, the `BOOLEAN` requires 1 bit, while the constrained `INTEGER` requires $\lceil \log_2(8) \rceil = 3$ bits. Hence, the entire structure can be encoded in just 4 bits, compared to at least 2 bytes in many other encoding schemes. This compactness is critical in bandwidth-constrained applications.

2.1.4 ASN.1 Control Notation (ACN)

While uPER and other standard encoding rules provide no control over the binary representation, certain protocols — notably ESA's PUS-C [17] — impose requirements that cannot be expressed within those rules. The ASN.1 Control Notation (ACN) [18] addresses this limitation by supplementing ASN.1 type definitions with explicit encoding annotations written in separate files.

ACN annotations govern details such as bit widths, byte order, alignment, padding, field ordering, and the encoding of length and choice determinants. ACN can be regarded as a generalization of uPER: in the absence of annotations, the encoding defaults to uPER, while annotations permit deviations to meet specific protocol requirements.

Listing 3 illustrates the ACN syntax. The ASN.1 definition declares a `SEQUENCE` with an `INTEGER` `length` field and a variable-length `OCTET STRING`. The ACN file overrides the default encoding by specifying that `length` is encoded as a 16-bit big-endian `INTEGER` and that the size of `data` is determined by the value of `length`.

```
Packet ::= SEQUENCE {
    length      INTEGER (0..255),
    data        OCTET STRING
}

Packet [] {
    length [size 16, encoding pos-int, endianness big],
    data   [size length]
}
```

Listing 3: ASN.1 type definition (top) and corresponding ACN annotation (bottom).

2.2 ASN1SCC

The ASN.1 Space Certifiable Compiler (ASN1SCC)[1] is an open-source compiler for ASN.1/ACN developed by the European Space Agency (ESA). It generates encoders and decoders in multiple programming languages from ASN.1 and ACN specifications. Supported targets include C, Ada, Scala, and Python, the latter being the focus of this thesis. ASN1SCC is specifically designed for space applications, where correctness and reliability are paramount.

2.2.1 Architecture

ASN1SCC follows a compiler pipeline illustrated in Figure 2.1. The compiler frontend parses and type-checks the ASN.1 and ACN input files, after which the backend generates language-specific code.

For each target language, the output consists of two components:

1. **Runtime Library**
   A fixed, language-specific library providing core encoding and decoding primitives. For Python, this includes the Bitstream component and basic type encoders and decoders.

[1] https://github.com/esa/asn1scc

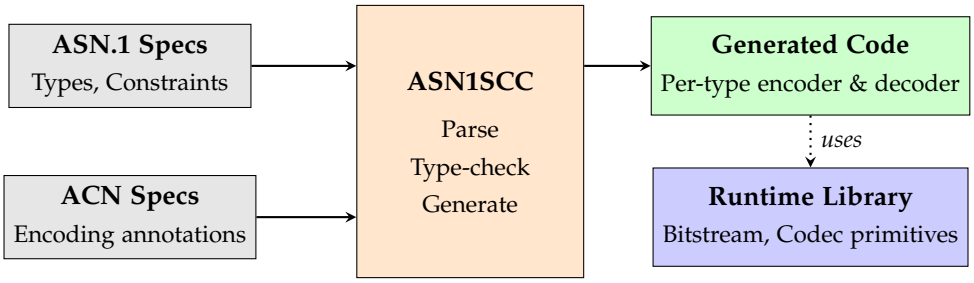

Figure 2.1: ASN1SCC architecture: ASN.1 and ACN specification files are compiled into per-type generated encoders and decoders. The runtime library is a fixed component bundled alongside the generated code.

2. **Generated Code**
   Type-specific encoder and decoder classes generated from the provided ASN.1/ACN specifications. This code uses the runtime library's primitives to implement the encoding and decoding logic for each defined type.

This separation allows the runtime library to be thoroughly tested and verified once, while the generated code handles the schema-specific logic.

2.3 Viper

Viper [13] is a program verification infrastructure that combines modular deductive verification with permission-based heap reasoning [16]. It provides its own intermediate verification language into which frontends encode their respective source languages.

In Viper's permission model, each heap location carries a permission: any positive fractional amount suffices for reading, while full permission of 1 is required for writing. Frontends such as Nagini for Python [7], Prusti for Rust [3], and Gobra for Go [19] translate their respective source languages into Viper programs.

Viper provides two verification backends: a symbolic execution backend (Silicon) and a verification condition generation backend (Carbon). Both backends discharge their proof obligations to SMT solvers using the SMT-LIB standard interface [4]; Silicon uses Z3 [6] by default.

2.4 Nagini

Nagini [7, 8] is an automatic verifier for statically typed Python programs. It translates Python source files annotated with formal specifications into Viper programs, which are subsequently verified by one of Viper's backends.

Verified code must be annotated with Mypy-compatible static type declarations [11], allowing Nagini to determine precise types for variables and expressions during translation.

Nagini's specification constructs take the form of Python function calls imported from the `nagini_contracts` package. This design turns specifications into syntactically valid Python code that can be type-checked by Mypy while having no semantic effect at runtime. The core constructs correspond closely to those of Viper.

Verification is modular: when checking a function, Nagini relies solely on the specifications of any impure functions it calls, not their implementations. Pure functions are an exception to this rule, as elaborated below.

```
def safe_divide(x: int, y: int) -> int:
    Requires(y != 0)
    Ensures(Result() == x // y)
    return x // y
```

Listing 4: A function with a precondition ruling out division by zero.

Method contracts are expressed via `Requires(expr)` and `Ensures(expr)`, which define preconditions and postconditions at the beginning of a function body (see Listing 4). Loop invariants are declared analogously using `Invariant(expr)`. Within postconditions, `Result()` denotes the return value, and `Old(expr)` evaluates an expression in the pre-state of the enclosing call. Quantified assertions are expressed using `Forall` and `Exists`.

```
def sum_to(n: int) -> int:
    Requires(n >= 0)
    Ensures(Result() == n*(n+1) // 2)

    res = 0
    i = 0
    while i < n:
        Invariant(0 <= i and i <= n)
        Invariant(res == i * (i+1) // 2)
        i += 1
        res += i
    return res
```

Listing 5: A summation function with a loop invariant enabling verification of its postcondition.

Since specifications must be side-effect-free, only pure expressions may appear in them. Functions annotated with `@Pure` satisfy this requirement: they are deterministic, free of observable side effects, and forbidden from modifying heap state. Consequently, they may be called within preconditions,

postconditions, and invariants. For pure functions, termination must be established explicitly using `Decreases(expr)`.

Pure functions are global by default: their definition is visible to callers. Annotating them with `@Opaque` restores modular treatment, restricting callers to the function's specification. When the function body is needed locally, `Reveal(func)` can be used to temporarily unfold the definition, making it available within the current scope. Listing 6 illustrates this difference.

```
@Pure
def pure_func() -> int:
    return 3

@Pure
@Opaque
def pure_opaque_func() -> int:
    return 3

Assert(pure_func() == 3)         # Can be proven
Assert(pure_opaque_func() == 3) # Cannot be proven
x = Reveal(pure_opaque_func())
Assert(x == 3)                   # Can be proven
```

Listing 6: Comparison between non-opaque and opaque pure functions. The assertion on the opaque function cannot be proven directly, but `Reveal` unfolds its definition locally, allowing the final assertion to be proven.

Following Viper's fractional permission model, heap permissions are expressed using `Acc(location, amount)`. When `amount` is omitted, full permissions are assumed. Permissions must be explicitly transferred at method boundaries: any method reading or writing a heap location must hold the corresponding permission, supplied by the caller via the precondition. Listing 7 shows an example for a class with a single field.

```
class MyClass:
    x: int

def read(m: MyClass) -> int:
    Requires(Acc(m.x, 1/2))
    Ensures(Acc(m.x, 1/2))
    return m.x

def incr(m: MyClass) -> None:
    Requires(Acc(m.x))
    Ensures(Acc(m.x))
    m.x += 1
```

Listing 7: Read and increment functions accessing the field of a class.

Predicates, declared with the `@Predicate` decorator, are named abstractions over arbitrary assertions, typically bundling access permissions and constraining field values. They serve as the primary mechanism for managing permissions at method boundaries: a method requiring access to an object's fields declares the corresponding predicate in its precondition, and the caller is responsible for holding and transferring it.

Access to the underlying permissions and assertions is obtained by unfolding the predicate via `Unfold`, which exchanges the predicate for its body. `Fold` performs the inverse operation. Listing 8 uses the same class as the previous example, but access permissions are conveyed via a predicate instead.

```
class MyClass:
    x: int

@Predicate
def pred(m: MyClass) -> bool:
    return Acc(m.x)

def func(m: MyClass) -> int:
    Requires(Acc(pred(m), 1/2))
    Ensures(Acc(pred(m), 1/2))
    Unfold(Acc(pred(m), 1/2))
    r = m.x
    Fold(Acc(pred(m), 1/2))
    return r

def incr(m: MyClass) -> None:
    Requires(Acc(pred(m)))
    Ensures(Acc(pred(m)))
    Unfold(pred(m))
    m.x += 1
    Fold(pred(m))
```

Listing 8: Read and increment functions accessing the field of a class using a predicate.

2.5 Stainless

Bucev et al. [5] employed Stainless [10] to formally verify the Scala backend of ASN1SCC, establishing functional correctness for a representative set of ASN.1 data types.

Stainless is a formal verification framework for Scala programs. As with Nagini, programs are annotated with preconditions, postconditions, and termination measures, and the resulting proof obligations are discharged by an SMT solver. To support imperative features, Stainless applies a preprocessing phase that rewrites mutations into functional equivalents: array writes and

field updates are translated into functional copies of the modified structure, and aliasing is restricted such that at most one reference to any mutable object exists within a given scope. Concurrency with a shared mutable heap is not supported. Consequently, no explicit permission model is required.

By contrast, the Nagini–Viper stack accommodates unrestricted aliasing by tracking read and write access to each heap location through fractional permissions.

Chapter 3

Verification of Runtime Library

The Python runtime library provides the foundation shared by all generated encoders and decoders of ASN1SCC. It consists of the `BitStream` data structure and the `Encoder` and `Decoder` base classes that build on it. The central verification goal throughout this work is round-trip correctness: encoding an ASN.1 value and subsequently decoding it must recover the original.

Figure 3.1 provides an overview of the component hierarchy. In this chapter, we focus on the `BitStream` and the `Encoder` / `Decoder` layer, while Chapter 4 covers the top layer of type-specific encoders and decoders. Since all generated encoders and decoders rely on this runtime library, establishing its correctness is a prerequisite for verifying the correctness of the generated code.

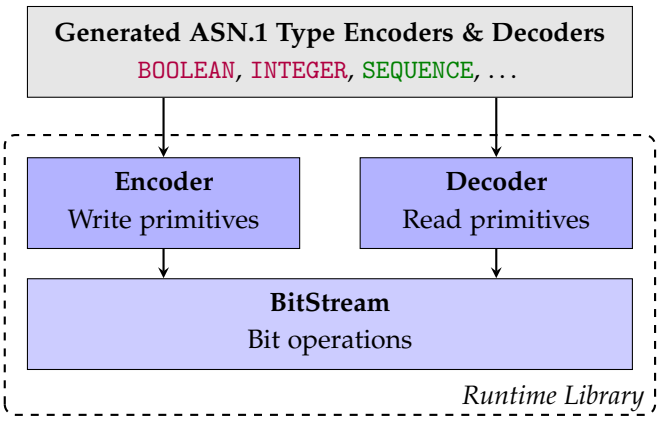

Figure 3.1: Component hierarchy of the Python implementation. The `BitStream` provides low-level bit operations on which the `Encoder` and `Decoder` implement the primitive operations. The generated ASN.1 type encoders and decoders rely on these primitives to perform their encoding and decoding.

Figure 3.2: Role of the `BitStream` class in the encoding / decoding pipeline.

In practice, the encoded `BitStream` contents are transmitted over a network to a remote client, which feeds them into its own `BitStream` and runs the decoder.

3.1 BitStream

The `BitStream` class is the shared buffer through which all encoding and decoding is mediated: encoders write the bit representation of each ASN.1 field to it sequentially, and decoders read them back in the same order (Figure 3.2).

```
def read_bit() -> bool:

def read_bits(bit_count: int) -> int:

def write_bit(bit: bool) -> None:

def write_bits(value: int, bit_count: int) -> None:
```

Listing 9: Main interface of the `BitStream` class.

Listing 9 shows the public interface of the `BitStream` class, which consists of two symmetric pairs: `write_bit` and `write_bits` for encoding, and `read_bit` and `read_bits` for decoding. The method `write_bits` encodes an integer value using a specified number of bits, while `read_bits` reconstructs the value by reading the same number of bits. The single-bit variants operate on one bit at a time. The core codec operations delegate to these methods, alongside additional helper functions such as `read_align_byte`. Figure 3.3 illustrates the usage of the `BitStream`.

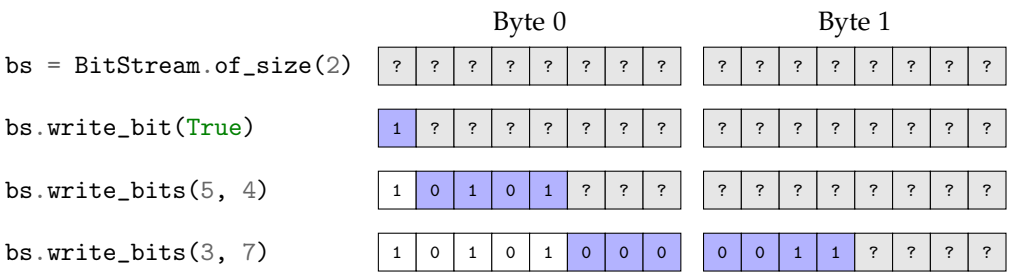

Figure 3.3: `BitStream` write operations and the resulting bytearray state after each call.

Internally, the `BitStream` stores data in a Python `bytearray` called `_buffer` and maintains the current bit offset using the integer fields `_current_bit` and `_current_byte`. These fields are exposed through the `current_used_bits` property. A complementary property, `remaining_bits`, exposes the number of bits still available.

```
current_used_bits = _current_byte * 8 + _current_bit
remaining_bits = len(_buffer) * 8 - current_used_bits
```

Write operations set bits in MSB-first order starting at this position and advance the offset accordingly. After encoding, the offset is reset so that reads consume bits in FIFO order. Both operations delegate through a chain of internal methods down to the bit-level operations `byte_set_bit` and `byte_read_bit`, as illustrated in Figure 3.4.

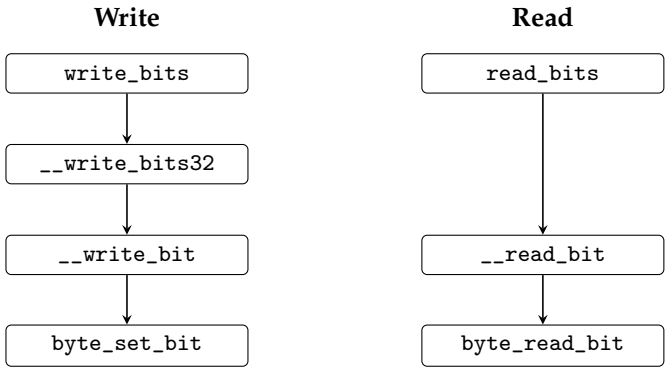

Figure 3.4: Implementation call chains for write and read operations in the `BitStream`.

The write chain includes one additional intermediate step, `__write_bits32`, which was introduced to make verification feasible. Listing 10 shows the entry point: `write_bits` splits the write into two 32-bit halves and delegates each to `__write_bits32`; the top-level invocation is limited to 64 bits.

```
def write_bits(value: int, bit_count: int) -> None:
    # ...
    if bit_count > 32:
        lower_count = bit_count - 32
        __write_bits32(value >> lower_count, 32)
        __write_bits32(value % (1 << lower_count), lower_count)
    else:
        __write_bits32(value, bit_count)
```

Listing 10: Implementation of `write_bits`, splitting into two 32-bit calls.

Verification

For verification, the pure getter function `buffer()` exposes the buffer state as a `PByteSeq` — a pure, immutable counterpart to `bytearray` — allowing it to be referenced in specifications without requiring heap permissions. It calls `ToByteSeq(_buffer)` internally.

The central verification challenge is reasoning about cross-byte operations: a write operation may span two or more bytes, requiring the proof to account for partial updates to the affected bytes while preserving all previously written bits (Figure 3.5). Propagating this low-level reasoning through every layer of the proof would be prohibitively expensive.

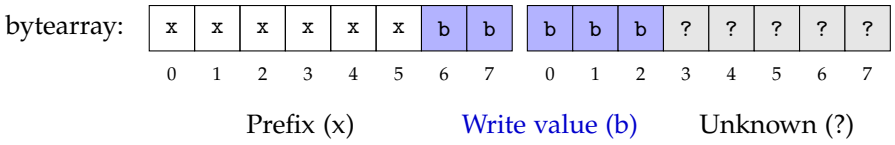

Figure 3.5: Write operation spanning byte boundaries

To address this, we structured the verification of the `BitStream` around three layers of abstraction shown in Figure 3.6:

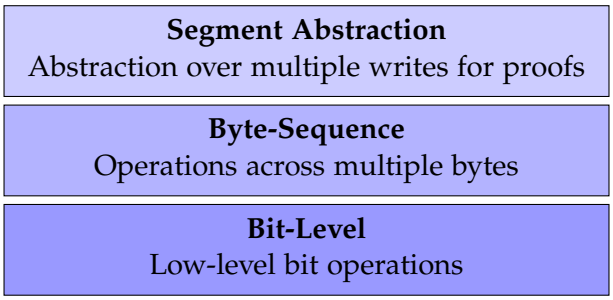

Figure 3.6: Three-layer architecture for the verification of the `BitStream` component.

1. **Bit-Level Layer**
   Low-level operations on individual bytes, including reading and writing single or multiple bits within a byte.
2. **Byte-Sequence Layer**
   Extends bit-level operations across multiple bytes, managing the underlying `bytearray` and tracking the current position.
3. **Segment Abstraction Layer**
   Used exclusively for verification, this layer records consecutive write operations as abstract value–length pairs, eliminating the need for bit-level reasoning at higher layers.

The lower two levels closely mirror the actual implementation, while the top layer has no counterpart in the runtime. Each layer is verified independently, with the resulting proofs composed bottom-up through separate lemmata.

3.1.1 Bit-Level Operations

At the lowest level, individual bits are read and written within Python integers representing bytes. The corresponding functions are shown in Listing 11.

```
@Pure
@Opaque
def byte_set_bit(byte: int, position: int, bit: bool) -> int:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < 8)
    Ensures(0 <= Result() and Result() <= 0xFF)
    if bit:
        return byte | (1 << (7 - position))
    else:
        return byte & ~(1 << (7 - position))

@Pure
@Opaque
def byte_read_bit(byte: int, position: int) -> bool:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < 8)
    return bool((byte >> (7 - position)) % 2)
```

Listing 11: Bit-level functions with specifications.

The first function, `byte_set_bit`, returns a copy of `byte` with the bit at `position` set or cleared, while `byte_read_bit` returns the bit at the same position as a boolean. Both use MSB-first indexing, where position 0 denotes the most significant bit. Additionally, the functions are marked `@Opaque`, restricting callers to their specifications rather than their implementations — the pure-opaque pattern discussed in section 6.1. Their specifications comprise preconditions constraining the permissible inputs and, for `byte_set_bit`, a postcondition bounding the output range.

Two additional pure functions with no runtime counterpart are introduced at this level: `byte_set_bits` and `byte_read_bits` (shown in Listing 12). These operate on multiple bits within a single byte integer and are used exclusively for proofs; their parameters are declared as `PInt`, a pure integer variant discussed in section 6.2. Their equivalence to the single-bit variants is established by separate lemmata.

```
@Pure
@Opaque
def byte_set_bits(
        byte: PInt, value: PInt,
        position: PInt, length: PInt) -> int:
    Requires(byte constraints)
    Ensures(0 <= Result() and Result() <= 0xFF)

    upper = byte_read_bits(byte, 0, position) << (8 - position)
    middle = value << (8 - position - length)
    lower = byte % (1 << (8 - position - length))
    return upper + middle + lower

@Pure
@Opaque
def byte_read_bits(
        byte: PInt, position: PInt,
        length: PInt) -> int:
    Requires(byte constraints)
    Ensures(0 <= Result() and Result() < (1 << length))

    return (byte >> (8 - position - length)) % (1 << length)
```

Listing 12: Multi-bit write and read operation for a single byte.

The body of `byte_set_bits` reconstructs the result from three additive components:

- `upper` preserves the bits before the write range (shifted toward the most significant bit),
- `middle` places the new value at the target position,
- `lower` preserves the bits after the write range.

This arithmetic decomposition avoids bitwise AND and OR entirely, which simplifies the SMT encoding discussed in section 3.1.5. The key properties established at this level are:

```
new_byte = byte_set_bits(byte, value, pos, length)

byte_read_bits(new_byte, pos, length) == value
byte_read_bits(new_byte, 0, pos) == byte_read_bits(byte, 0, pos)
```

Rather than encoding these directly as postconditions of `byte_set_bits`, they are established as separate lemmata, a structural pattern used throughout the verification and discussed further in section 6.1.

3.1.2 Byte-Sequence Operations

The byte-sequence layer extends the bit-level operations to the full `bytearray`. As at the bit level, it introduces four pure specification functions operating on a `PByteSeq`:

`byteseq_set_bit`	`byteseq_read_bit`
`byteseq_set_bits`	`byteseq_read_bits`

Listing 13 shows the two write (set) variants.

```
@Pure
@Opaque
def byteseq_set_bit(
        seq: PByteSeq, bit: PBool,
        position: PInt) -> PByteSeq:
    Requires(0 <= position and position < len(seq) * 8)
    Ensures(len(seq) == len(Result()))

    byte_position = position // 8
    bit_position = position % 8
    return seq.update(byte_position,
            byte_set_bit(seq[byte_position], bit, bit_position))

@Pure
@Opaque
def byteseq_set_bits(
        seq: PByteSeq, value: PInt,
        position: PInt, length: PInt) -> PByteSeq:
    Requires(...)
    Decreases(length)
    Ensures(len(seq) == len(Result()))

    if length == 0:
        return seq

    rec = byteseq_set_bits(seq, value >> 1, position, length - 1)
    bit = bool(value % 2)
    return byteseq_set_bit(rec, bit, position + length - 1)
```

Listing 13: Pure byte-sequence write functions.

The single-bit variant, `byteseq_set_bit`, decomposes the flat bit position into a byte index and an intra-byte offset before delegating to `byte_set_bit`.

The multi-bit counterpart, `byteseq_set_bits`, is defined recursively: each call extracts the least significant bit and places it at the final position of the range, recursing until the length reaches zero. The read counterparts are defined analogously.

The key properties to establish at this level are analogous to those of the bit-level layer, but extended across byte boundaries. A write spanning a byte boundary must leave all previously written bits unchanged; we capture this property using the helper `byteseq_equal_until` (Listing 14). This function checks that two byte sequences agree on all bits up to a given position: full bytes are compared as a slice, and if the position falls mid-byte, the leading bits of that partial byte are compared using `byte_read_bits`.

```
def byteseq_equal_until(
        b1: PByteSeq, b2: PByteSeq,
        end: PInt) -> bool:
    Requires(0 <= end and
             end <= len(b1) * 8 and
             end <= len(b2) * 8)

    full_bytes = end // 8
    bit_position = end % 8

    is_eq = b1.take(full_bytes) == b2.take(full_bytes)

    if is_eq and bit_position > 0:
        byte1 = b1[full_bytes]
        byte2 = b2[full_bytes]
        is_eq = (byte_read_bits(byte1, 0, bit_position) ==
                 byte_read_bits(byte2, 0, bit_position))

    return is_eq
```

Listing 14: Bit-precise prefix equality over two byte sequences.

Both the prefix preservation and round-trip properties are established by `lemma_byteseq_set_bits` (Listing 15), proved by induction over the byte sequence.

```
def lemma_byteseq_set_bits(
        byteseq: PByteSeq, value: int,
        position: int, length: int) -> bool:
    # ...
    Ensures(byteseq_equal_until(
                byteseq_set_bits(byteseq, value, position, length),
                byteseq, position))
    Ensures(byteseq_read_bits(
                byteseq_set_bits(byteseq, value, position, length),
                position, length) == value)
```

Listing 15: Key lemma for byte-sequence writes.

With these properties in place, `__write_bit` is the core implementation operation, whose postconditions establish the key link between the implementation and the specification: the bit offset advances by one, and the buffer state is updated according to `byteseq_set_bit`, as shown in Listing 16.

```
def __write_bit(self, bit: bool) -> None:
    # ...
    Ensures(self.current_used_bits ==
            Old(self.current_used_bits) + 1)
    Ensures(self.buffer() == Old(byteseq_set_bit(self.buffer(),
                                 bit, self.current_used_bits)))
```

Listing 16: Postconditions of `__write_bit`.

Multi-bit writes are implemented by `__write_bits32`, which composes multiple calls to `__write_bit`. The postconditions of `__write_bits32` mirror those of `__write_bit`, but at the level of `byteseq_set_bits`, as shown in Listing 17.

```
def __write_bits32(self, value: int, bit_count: int) -> None:
    # ...
    Ensures(self.current_used_bits ==
            Old(self.current_used_bits) + bit_count)
    Ensures(self.buffer() == byteseq_set_bits(
                             Old(self.buffer()), value,
                             Old(self.current_used_bits),
                             bit_count))

    while i < bit_count:
        # ...
        Invariant(self.current_used_bits ==
                  Old(self.current_used_bits) + i)
        Invariant(ghost_value == value >> (bit_count - i))
        Invariant(self.buffer() == byteseq_set_bits(...,
                                   ghost_value, ..., i))

        bit = bool((value >> (bit_count - 1 - i)) % 2)
        self.__write_bit(bit)

        ghost_value = (ghost_value << 1) + bit
        Reveal(byteseq_set_bits(..., ghost_value, ..., i + 1))
        i = i + 1
```

Listing 17: Postconditions, loop invariant, and body of `__write_bits32`.

The listing shows a representative subset of the loop invariants. Among them is an invariant for `ghost_value = value >> (bit_count - i)`. That

is, `ghost_value` represents the upper i bits of `value` that have already been written. This ghost state is necessary because the buffer invariant refers to the partially written value, which cannot be expressed directly in terms of `value` and i alone.

Each iteration extracts the next bit from `value` in MSB-first order and forwards it to `__write_bit`. The ghost variable is then updated by appending the written bit. Finally, `Reveal` explicitly unfolds the opaque definition of `byteseq_set_bits` for $i + 1$ bits, enabling the verifier to connect consecutive loop states and re-establish the buffer invariant.

Closing the write chain, `write_bits` invokes `__write_bits32` twice — once for the upper 32 bits and once for the remainder — and applies the combine lemma displayed in Listing 18.

```
@Pure
@Opaque
def lemma_byteseq_set_bits_combine(
        seq: PByteSeq, value: int,
        upper_length: int, lower_length: int,
        position: int) -> bool:
    Requires(...)
    Decreases(lower_length)
    Ensures(two sequential, partial writes == full write)

    if lower_length == 0:
        return True

    combined_length = upper_length + lower_length
    upper_value = value >> lower_length
    lower_value = value % (1 << lower_length)
    intermediate = byteseq_set_bits(seq, upper_value,
                                    position, upper_length)

    lhs_full = Reveal(byteseq_set_bits(
                        intermediate, lower_value,
                        position + upper_length, lower_length))
    rhs_full = Reveal(byteseq_set_bits(
                        seq, value, position, combined_length))

    rec_lemma = lemma_byteseq_set_bits_combine(
                    seq, value >> 1, upper_length,
                    lower_length - 1, position)

    return lhs_full == rhs_full
```

Listing 18: Combine lemma for consecutive byte-sequence writes.

This lemma proves that the two sequential, partial writes are equivalent to

a single call to `byteseq_set_bits` with the full value and length. Splitting a write operation of 64 bits into two halves is necessary, because directly verifying a 64-bit write does not terminate within practical time limits.

The read side is simpler: `__read_bit` reads a single bit from `_buffer` and `read_bits` composes multiple such reads directly, without requiring the additional split used for the writes.

3.1.3 Segment Abstraction

Each write operation to the `BitStream` updates both the current bit position and potentially multiple bytes in the underlying array. During the decoding phase, the `BitStream` is read in the same order in which the bits were written. Reasoning about these low-level changes throughout the encoding and decoding would be cumbersome and prohibitively expensive for the verifier. To address this issue, we introduce a segment abstraction that is used exclusively for verification. `Segment` is implemented as a Nagini algebraic data type (ADT), giving it structural equality and immutability. Listing 19 presents a simplified form for readability.

```
class Segment:
    length: int
    value: int

@Pure
def segment_invariant(seg: Segment) -> bool:
    return (0 <= seg.length and seg.length <= 64 and
            0 <= seg.value and seg.value < (1 << seg.length))
```

Listing 19: Definition of `Segment` (simplified from the Nagini ADT form) and invariant.

A `Segment` stores only the length and value of a write operation, thereby abstracting away the specific byte-level representation. Additionally, all functions operating on segments require and ensure that the value of a segment lies within the range $[0, 2^{length} - 1]$ through a helper function, `segment_invariant`. The length of a `Segment` is limited to 64, mirroring the maximum number of bits written by a single call to `write_bits`.

Segments are consecutive and non-overlapping by construction, reflecting the sequential write behavior of the encoders. Hence, the total bit length of a sequence of segments is equal to the sum of lengths, which greatly simplifies verification. A recursive function, `segments_total_length`, is defined accordingly, as illustrated in Listing 20.

The key invariant linking the ghost segment sequence to the actual buffer is `segments_contained`, shown in Listing 21. It recursively checks that each segment's value can be read back from the buffer at the correct bit position

```
@Pure
def segments_total_length(segments: PSeq[Segment]) -> int:
    Requires(Forall(segments, lambda seg: segment_invariant(seg)))
    Decreases(len(segments))

    if len(segments) == 0:
        return 0

    last_idx = len(segments) - 1
    rec_segments = segments.take(last_idx)
    last_seg = segments[last_idx]
    return segments_total_length(rec_segments) + last_seg.length
```

Listing 20: Recursive function for computing the total bit length of a segment sequence.

using `byteseq_read_bits`. The companion predicate, `segments_invariant`, bundles `segments_contained`, the per-element `segment_invariant`, and the bounds condition into a single function that is used as class invariant of the `BitStream` class.

```
@Pure
def segments_contained(byteseq: PByteSeq,
        segments: PSeq[Segment]) -> bool:
    Requires(Forall(segments, lambda seg: segment_invariant(seg)))
    Requires(segments_total_length(segments) <= len(byteseq) * 8)
    Decreases(len(segments))

    if len(segments) == 0:
        return True

    last_idx = len(segments) - 1
    rec_segments = segments.take(last_idx)
    rec_length = segments_total_length(rec_segments)
    last_seg = segments[last_idx]

    contained_bits = byteseq_read_bits(byteseq, rec_length,
                        last_seg.length)
    contained = contained_bits == last_seg.value
    rec = segments_contained(byteseq, rec_segments)
    return contained and rec

@Pure
def segments_invariant(byteseq: PByteSeq,
        segments: PSeq[Segment]) -> bool:
    return (Forall(segments, lambda seg: segment_invariant(seg)) and
            segments_total_length(segments) <= len(byteseq) * 8 and
            segments_contained(byteseq, segments))
```

Listing 21: `segments_contained` and `segments_invariant`.

Higher-layer proofs therefore never reason about the buffer content directly: a call to `write_bits(v, n)` appends a segment to the ghost sequence, and a subsequent `read_bits(n)` returns exactly that value, so round-trip correctness reduces to an assertion over segment sequences alone (Figure 3.7).

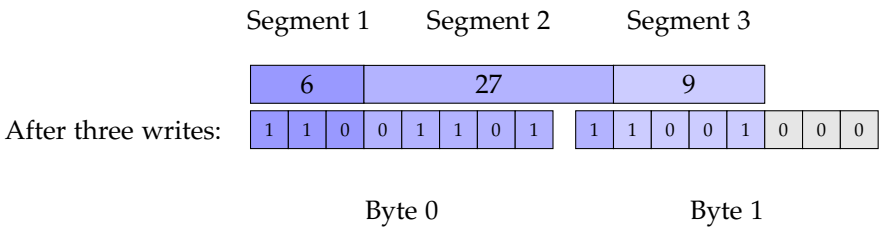

Figure 3.7: Segment abstraction tracking three write operations. The upper layer shows the segments, while the lower layer shows the corresponding bit-level representation.

3.1.4 BitStream Class

The `BitStream` class ties together all three layers under a single verified interface. It holds the `bytearray` buffer, the `_current_bit` and `_current_byte` position fields, and the ghost fields `_segments` and `_segments_read_index`. Its invariant is split across two Nagini predicates, shown in Listings 22 and 23.

```
@Pure
def position_invariant(bit_position: int, byte_position: int,
        buf_length: int) -> bool:
    return (bit_position >= 0 and bit_position < 8 and
            byte_position >= 0 and
            ((byte_position < buf_length) or
             (bit_position == 0 and byte_position == buf_length)))

@Predicate
def bitstream_invariant(self) -> bool:
    return (Acc(self._current_bit) and
            Acc(self._current_byte) and
            Acc(self._buffer) and
            Acc(bytearray_pred(self._buffer)) and
            position_invariant(self._current_bit,
                self._current_byte, len(self._buffer)))
```

Listing 22: Predicate and helper function governing the concrete fields and position bounds.

The first predicate, `bitstream_invariant`, owns the concrete fields and enforces position bounds via `position_invariant`: the byte position may equal `buf_length` only when the intra-byte bit position is zero, capturing the state in which the stream is exactly full. The second predicate,

```
@Predicate
def segments_predicate(self, byteseq: PByteSeq) -> bool:
    return (Acc(self._segments) and
            segments_invariant(byteseq, self._segments) and
            Acc(self._segments_read_index) and
            0 <= self._segments_read_index and
            self._segments_read_index <= len(self._segments))
```

Listing 23: Predicate governing the ghost segment sequence and read index.

`segments_predicate`, owns the ghost segment sequence and ties it to the buffer via `segments_invariant`, ensuring that the ghost state faithfully represents the data written.

The ghost fields are exposed to callers through the pure properties shown in Listing 24. Both require the caller to hold `bitstream_invariant` (needed to evaluate `buffer()`) and `segments_predicate`. As pure functions, the properties call `Unfold` to access the field enclosed in the predicate and return it directly.

```
@property
def segments(self) -> PSeq[Segment]:
    Requires(self.bitstream_invariant())
    Requires(self.segments_predicate(self.buffer()))
    Unfold(self.segments_predicate(self.buffer()))
    return self._segments

@property
def segments_read_index(self) -> int:
    Requires(self.bitstream_invariant())
    Requires(self.segments_predicate(self.buffer()))
    Unfold(self.segments_predicate(self.buffer()))
    return self._segments_read_index
```

Listing 24: Pure properties exposing the ghost segment sequence and read index.

Both `write_bit` and `write_bits` append exactly one segment per call, recording its length and value regardless of how many bytes the operation spans, and maintain `segments_invariant`. Symmetrically, a read operation returns the value of the segment at `segments_read_index` and advances it by one. The precondition for reading requires the requested bit count to match the length of the current segment exactly. This constraint is always satisfied in a round trip, because the decoder calls `read_bits` with the same lengths that the encoder passed to `write_bits`, so the read and write sequences are aligned by construction. Higher-layer proofs access the ghost state exclusively through these properties, allowing all specifications to be expressed in terms of segment sequences rather than buffer contents.

3.1.5 Design Decisions

A key challenge in implementing the bit-level operations was representing bit shifts efficiently in Nagini. In contrast to many other programming languages, Python integers are unbounded, so Nagini represents them as SMT-LIB integers rather than fixed-width bitvectors, as Stainless does for Scala. While Nagini supports bitwise AND and OR through SMT-LIB bitvector operations, it did not initially support shift operations.

The natural approach is to implement shifts via `shlBVInt`, which translates to a `bvshl` call in SMT-LIB, converting between integers and bitvectors using `toBVInt` and `fromBVInt`, as presented in Listing 25.

```
function __prim__int___lshift__(self: Int, other: Int): Int
    requires ...
{
    self >= 0 ?
        fromBVInt(shlBVInt(toBVInt(self), toBVInt(other))) :
        -fromBVInt(shlBVInt(toBVInt(-self), toBVInt(other)))
}
```

Listing 25: Bitvector-based Viper encoding of left shift.

Nagini is invoked with the argument `--int-bitops-size`, which controls the bitvector width used by `toBVInt` and `fromBVInt`. The bitvector shift approach requires this width to accommodate the shift amount, which can be up to 64 bits. However, increasing `--int-bitops-size` also widens the bitvectors used for all AND and OR operations in the same verification run, including the byte-level operations that only require 8-bit bitvectors. This coupling makes the bitvector approach globally expensive.

Instead, we encode shifts as integer multiplication using a case distinction over all shift amounts up to 64, as illustrated in Listing 26. A case distinction is preferred over a recursive power function because the SMT solver can retrieve the multiplication factor directly without unrolling. With this approach, `--int-bitops-size` can be kept at 8 (9 with the sign bit), so AND and OR operations remain on small, byte-sized bitvectors. Shift amounts larger than 64 fall back to the bitvector variant, so code requiring them will incur the same performance penalty. In practice, the implementation never shifts by more than 64 bits, so this restriction does not apply.

Impact

To evaluate the impact, we benchmarked the two approaches on the read functions shown in Listing 27. Figure 3.8 demonstrates that verification time increases drastically with larger bit-operation sizes under the bitvector approach, while the integer arithmetic variant remains constant.

```
function __prim__int___lshift__(self: Int, other: Int): Int
    requires ...
{
    0 <= other <= 64 ? self * __shift_factor64(other) :
    0 <= self ?
            fromBVInt(shlBVInt(toBVInt(self), toBVInt(other))) :
            -fromBVInt(shlBVInt(toBVInt(-self), toBVInt(other)))
}

function __shift_factor64(amount: Int): Int
    requires 0 <= amount && amount <= 64
{
    amount <= 32 ? __shift_factor32(amount) :
            __shift_factor32(amount - 32) * __shift_factor32(32)
}

function __shift_factor32(amount: Int): Int
    requires 0 <= amount && amount <= 32
{
    amount <= 8 ?  __shift_factor8(amount) :
    amount <= 16 ? __shift_factor8(amount - 8) * 256 :
    // ...
}

function __shift_factor8(amount: Int): Int
    requires 0 <= amount && amount <= 8
{
    amount == 0 ? 1 :
    amount == 1 ? 2 :
    // ...
}
```

Listing 26: Integer-arithmetic encoding of a left shift in Viper, with a fallback to the bitvector variant for shift amounts exceeding 64 bits.

```
def byte_read_bit(byte: PInt, position: PInt) -> bool:

def byteseq_read_bit(byteseq: PByteSeq,
    position: PInt) -> bool:

def byteseq_read_bits(byteseq: PByteSeq,
    position: PInt, length: PInt) -> int:
```

Listing 27: Functions used in the read-bit benchmark.

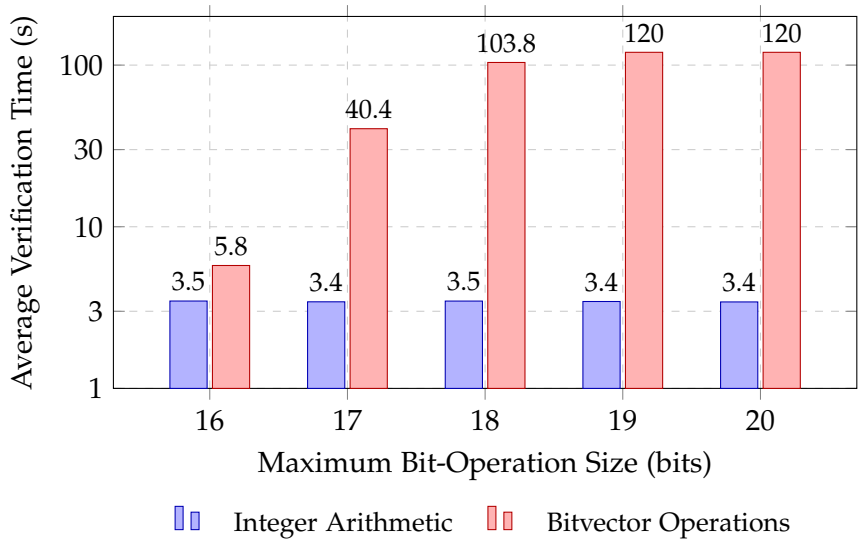

Figure 3.8: Average verification time for the read-bit test case (Listing 72 on page 88) comparing integer arithmetic with native SMT-LIB bitvector operations, across different maximum shift amounts. A timeout of 120 seconds was applied.

The effect is even more pronounced for the write functions (Listing 28), where the bitvector variant already exceeds the 120-second timeout for a bit-operation size of 13, as depicted in Figure 3.9.

```
def byte_set_bit(byte: PInt, bit: bool, position: PInt) -> int:

def byteseq_set_bit(byteseq: PByteSeq,
    bit: PBool, position: PInt) -> PByteSeq:

def byteseq_set_bits(byteseq: PByteSeq,
    value: PInt, position: PInt, length: PInt) -> PByteSeq:
```

Listing 28: Functions used in the set-bit benchmark.

One anomaly in the set-bit results is that a maximum bit-operation size of 12 bits consistently yields shorter verification times than 11 bits. The underlying cause is not apparent from the data and warrants further investigation. The bit-operation sizes in these benchmarks were chosen to illustrate the scaling behavior. The actual implementation uses a maximum of 64 bits.

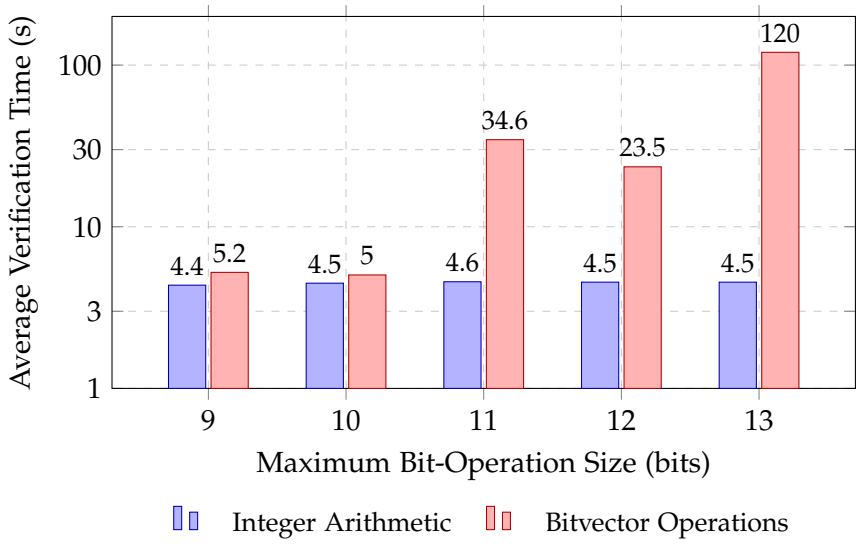

Figure 3.9: Average verification time for the set-bit test case (Listing 73 on page 89) comparing integer arithmetic with native SMT-LIB bitvector operations, across different maximum shift amounts. A timeout of 120 seconds was applied.

3.2 uPER Codec Methods

Rather than interacting with the `BitStream` directly, generated encoders and decoders invoke intermediary methods provided by the `Encoder` and `Decoder` classes. Each class wraps a `BitStream` instance and exposes codec methods with segment-level postconditions. Both classes share the `codec_predicate`, shown in Listing 29, which bundles the field access permission alongside the `BitStream`'s predicates, `bitstream_invariant` and `segments_predicate`, under a single, unified name.

```
@Predicate
def codec_predicate(self) -> bool:
    return (Acc(self._bitstream) and
            self._bitstream.bitstream_invariant() and
            self._bitstream.segments_predicate(
                self._bitstream.buffer()))
```

Listing 29: Shared predicate of the `Encoder` and `Decoder` classes.

On top of this shared predicate, each class defines an additional pure invariant relating the segment sequence to the current bit position. The `Encoder`'s `write_invariant` (Listing 30) asserts that the total length of all segments written so far equals `current_used_bits`. Symmetrically, the `Decoder`'s `read_invariant` (Listing 31) asserts that the total length of the segments up to `segments_read_index` equals `current_used_bits`.

```
@Pure
def write_invariant(self) -> bool:
    Requires(self.codec_predicate())
    all = self.segments
    return segments_total_length(all) == self.current_used_bits
```

Listing 30: Write invariant of the `Encoder`.

```
@Pure
def read_invariant(self) -> bool:
    Requires(self.codec_predicate())
    prefix = self.segments.take(self.segments_read_index)
    return segments_total_length(prefix) == self.current_used_bits
```

Listing 31: Read invariant of the `Decoder`.

This symmetry ensures that the segment-level postconditions of encoding and decoding methods align by construction, facilitating the round-trip proofs presented in Chapter 4. Most methods are single-segment wrappers: the encoder delegates to `write_bit` or `write_bits` and forwards the resulting postconditions over segments, while the decoder performs the corresponding operations with `read_bit` or `read_bits`. The methods verified in this thesis constitute a representative subset; the Python ASN1SCC backend relies on further codec methods that are not covered here. Among the verified cases, two warrant closer examination.

3.2.1 Integer Encoding and Decoding

The method `encode_integer` implements the uPER constraint-exploitation scheme: given a value $v \in [min, max]$, it calculates and encodes the offset $v - min$ rather than the value directly, using the minimum number of bits required to represent the entire range. The bit width is computed as `(max_val - min_val).bit_length()`, which in Python corresponds to $\lceil \log_2(max - min + 1) \rceil$.

```
def encode_integer(self, value: int,
                   min_val: int, max_val: int,) -> EncodeResult:
    # ...
    bits_needed = (max_val - min_val).bit_length()
    self._bitstream.write_bits(value - min_val, bits_needed)
    # ...
```

Listing 32: Excerpt of `encode_integer`.

The symmetric method `decode_integer` reads a single segment, increments `segments_read_index` by one, and recovers the original value by adding *min* back to the stored offset.

3.2.2 Octet String Encoding and Decoding

The method `encode_octet_string_no_length` produces one segment per byte, unlike the integer methods which produce a single segment. It writes n bytes from a `bytearray` to the `BitStream` without a length prefix, and its postcondition uses the helper `segments_from_byteseq_full` to describe the resulting sequence. This helper function converts the first n bytes of a `PByteSeq` into a sequence of n segments of length 8, each carrying the corresponding byte value.

The actual work is performed by another method, `append_byte_array` (Listing 33), which iterates over the bytes and calls `append_byte` for each one. Its loop invariant tracks the segments appended so far: after i iterations, `segments` equals the original sequence extended by the first i bytes of `data` as segments. The `Reveal` call is necessary to unfold the opaque definition of `segments_from_byteseq_full` at each step, connecting the result after $i + 1$ iterations to the result after i.

```
def append_byte_array(self, data: bytearray, num_bytes: int):
    Requires(self.codec_predicate() and self.write_invariant())
    Requires(0 <= num_bytes and num_bytes <= len(data))
    Requires(self.remaining_bits >= num_bytes * 8)
    # ...
    Ensures(self.segments is Old(self.segments)
        + segments_from_byteseq_full(
            ToByteSeq(data).take(num_bytes)))
    # ...
    i = 0
    while i < num_bytes:
        # ...
        Invariant(0 <= i and i <= num_bytes)
        Invariant(self.remaining_bits >= (num_bytes - i) * 8)
        Invariant(self.segments == Old(self.segments)
            + Reveal(segments_from_byteseq_full(
                     ToByteSeq(data).take(i))))

        self.append_byte(data[i])
        i += 1
```

Listing 33: Bytearray encode helper.

The method `encode_octet_string_no_length`, shown in Listing 34, is then a thin wrapper that delegates to `append_byte_array` while preserving the same postcondition.

Following the established pattern, the symmetric `read_byte_array` advances the read index by n and guarantees that each byte of the returned `bytearray` equals the value of the corresponding segment. A companion method,

```
def encode_octet_string_no_length(
        self, data: bytearray, num_bytes: int):
    Requires(self.codec_predicate() and self.write_invariant())
    Requires(0 <= num_bytes and num_bytes <= len(data))
    Requires(self.remaining_bits >= num_bytes * 8)
    # ...
    Ensures(self.segments is Old(self.segments)
        + segments_from_byteseq_full(
            ToByteSeq(data).take(num_bytes)))
    # ...
    self.append_byte_array(data, num_bytes)
```

Listing 34: Octet string encoder.

`encode_octet_string_no_length_vec`, accepts a `List[int]` instead of a `bytearray`. It establishes the identical postcondition by converting its input and delegating to `encode_octet_string_no_length`.

3.3 Error Handling

The verified properties of the runtime library hold along the happy path: the preconditions of all methods assume that the buffer has sufficient capacity and that the input data is well-formed. Nevertheless, the runtime implementation includes error-handling infrastructure for cases in which these conditions are violated.

```
@dataclass(frozen=True)
class EncodeResult:
    success: bool
    error_code: ErrorCode
    bits_encoded: int = 0
    error_message: Optional[str] = None

@dataclass(frozen=True)
class DecodeResult(Generic[TDVal]):
    success: bool
    error_code: ErrorCode
    decoded_value: Optional[TDVal] = None
    bits_consumed: int = 0
    error_message: Optional[str] = None
```

Listing 35: Definitions of `EncodeResult` and `DecodeResult`.

`BitStream` operations raise a `BitStreamError` when an access would exceed the buffer boundary. The `Encoder` and `Decoder` methods wrap their return values in `EncodeResult` and `DecodeResult`, respectively (Listing 35). Both are frozen dataclasses — whose fields are immutable once initialised —

that carry a `success` flag, an `ErrorCode` (an `IntEnum` enumerating the possible failure modes), and additional context fields. `DecodeResult` is generic, parameterised by the type of the decoded value.

The failure paths through this infrastructure (the raised `BitStreamError` and the `success == False` branches of `EncodeResult` and `DecodeResult`) are not formally specified: the current preconditions ensure the verifier only ever reasons about the successful case. A detailed discussion of what would be required to extend verification to cover error cases is provided in section 4.4.

Listing 36 shows the error guards in `BitStream.write_bits` while Listing 37 illustrates how `Encoder.append_bit` catches a `BitStreamError` and wraps it in an `EncodeResult`.

```
def write_bits(self, value: int, bit_count: int) -> None:
    # ...
    if bit_count < 0 or bit_count > 64:
        raise BitStreamError(...)
    if self.remaining_bits < bit_count:
        raise BitStreamError(...)
    if value < 0 or value >= (1 << bit_count):
        raise BitStreamError(...)
    # ...
```

Listing 36: Error guards in `BitStream.write_bits`.

```
def append_bit(self, bit_value: bool) -> EncodeResult:
    # ...
    try:
        self._bitstream.write_bits(bit_value, 1)
        return EncodeResult(
                success=True,
                error_code=ENCODE_OK,
                bits_encoded=1
            )
    except BitStreamError as e:
        return EncodeResult(
            success=False,
            error_code=ERROR_INVALID_VALUE,
            error_message=str(e)
        )
```

Listing 37: Error wrapping in `Encoder.append_bit`.

Chapter 4

Verification of Dynamically Generated Classes

The previous chapter established the runtime library, including the verified `BitStream` with its segment abstraction, and the `Encoder` and `Decoder` classes, whose methods carry segment-level postconditions. This chapter focuses on the top layer of the verification architecture: the classes generated by ASN1SCC for each ASN.1 type definition.

```
@dataclass
class GeneratedClass:
    # ... field declarations

    def is_constraint_valid(self) -> Asn1ConstraintValidResult:
        # ...

    def encode(self, codec: UPEREncoder) -> None:
        # ...

    @classmethod
    def decode(cls, codec: UPERDecoder) -> 'GeneratedClass':
        # ...
```

Listing 38: Structure of a generated ASN.1 dataclass.

For most types, ASN1SCC generates a dataclass whose fields correspond to the ASN.1 fields of that type, along with three methods, as outlined in Listing 38. The `INTEGER` and `ENUMERATED` types deviate from this structure, as described in section 4.2. The `is_constraint_valid` method checks any schema-defined constraints, such as integer bounds or string length limits, returning an `Asn1ConstraintValidResult` (Listing 39). Since the dataclass `Asn1ConstraintValidResult` is annotated as `frozen`, its instances are immutable and no access permissions to its fields are required. The

complementary `encode` and `decode` methods implement the codec for that type. Both methods delegate exclusively to the runtime library's `Encoder` and `Decoder` methods described in section 3.2 and, for composite types, to the `encode` and `decode` methods of their constituent fields. The central verification goal remains round-trip correctness: for any value that satisfies `is_constraint_valid`, decoding the encoded value must recover the original value.

```
@dataclass(frozen=True)
class Asn1ConstraintValidResult:
    is_valid: bool
    error_code: int = 0
    message: str = ""
```

Listing 39: The frozen dataclass returned by `is_constraint_valid`.

Because the generated structure is uniform across all types, verification follows a common pattern. First, a set of ghost functions and lemmata is introduced into every generated class to serve as the interface between per-type proofs and the runtime library. This pattern is then applied systematically to each supported ASN.1 type.

4.1 Auxiliary Functions

The situation here mirrors the elevation step taken in the runtime library, where the `Encoder` and `Decoder` classes were introduced to lift the segment abstraction from the `BitStream` to the codec level (see section 3.2). The same pattern recurs at the data class level: a uniform set of auxiliary methods and lemmata is added to every generated class to bridge this gap. Table 4.1 provides an overview of their roles and Listing 40 shows their signatures; the remainder of this section describes each function in turn.

Function	Role
`class_predicate`	Field ownership
`is_constraint_valid_pure`	Pure constraint check
`segments_of(self)`	Segment sequence of the encoding
`segments_valid(suffix)`	Checks suffix is a valid encoding
`segments_count(suffix)`	Segments consumed by decode
`segments_eq_lemma`	Injectivity of `segments_of`

Table 4.1: Auxiliary functions added to every generated class and their roles in the verification.

```
@Predicate
def class_predicate(self) -> bool:

@Pure
def is_constraint_valid_pure(self) -> bool:
    Requires(Rd(self.class_predicate()))

@Pure
@staticmethod
def segments_of(val: 'GeneratedClass') -> PSeq[Segment]:
    Requires(Rd(val.class_predicate()))
    Requires(val.is_constraint_valid_pure())
    Ensures(Forall(Result(), lambda seg: segment_invariant(seg)))
    Ensures(GeneratedClass.segments_valid(Result()))

@Pure
@staticmethod
def segments_valid(segments: PSeq[Segment]) -> bool:
    Requires(Forall(segments, lambda seg: segment_invariant(seg)))

@Pure
@staticmethod
def segments_count(segments: PSeq[Segment]) -> int:
    Requires(Forall(segments, lambda seg: segment_invariant(seg)))
    Requires(GeneratedClass.segments_valid(segments))

@staticmethod
def segments_eq_lemma(
        left: 'GeneratedClass',
        right: 'GeneratedClass') -> bool:
    Requires(GeneratedClass.segments_of(left) ==
             GeneratedClass.segments_of(right))
    Ensures(left == right)
```

Listing 40: Signatures of the auxiliary functions.

Following the standard Nagini pattern, a predicate `class_predicate` bundles field ownership for use at method boundaries. Since specifications must be side-effect-free, `is_constraint_valid_pure` provides a pure counterpart to the generated `is_constraint_valid` method. The latter is not pure because it constructs a new `Asn1ConstraintValidResult` instance. It appears as a precondition of `encode`, restricting encoding to values that satisfy the schema constraints, and as a postcondition of `decode`, re-establishing the same guarantee on the decoded result.

The more substantial additions concern the segment abstraction. We introduce `segments_of`, a pure ghost function that returns the sequence of segments that would result from encoding the object. This function provides the postconditions of `encode` and `decode` with a shared, specification-friendly

handle on the encoding result. The companion functions `segments_valid` and `segments_count` operate on the remaining suffix of the decoder's segment sequence: the former checks that the suffix starts with a valid encoding for the type, while the latter computes how many segments a call to `decode` would consume from that suffix. Any segments beyond those are not considered.

The lemma `segments_eq_lemma` establishes that `segments_of` is injective: two objects whose segment sequences are equal must themselves be equal. This lemma provides the central composability result and closes the round-trip argument. The postcondition of `decode` guarantees that the decoded result has the same segment sequence as the original value; injectivity then implies that the two objects are identical.

Listings 41 and 42 show the preconditions and postconditions of the generated `encode` and `decode` methods that are central to the round-trip proof. Both methods carry additional specifications — such as permissions for `codec_predicate` — that are omitted here for clarity. The first precondition of `encode` checks that the codec has sufficient remaining capacity for the encoding; the exact bound is type-specific and is filled in by the code generator.

```
def encode(self, codec: UPEREncoder) -> None:
    Requires(codec.remaining_bits >= required bits)
    Requires(self.is_constraint_valid_pure())
    # ...
    Ensures(codec.remaining_bits ==
            Old(codec.remaining_bits) - required bits)
    Ensures(codec.segments is Old(codec.segments) +
                              segments_of(self))
```

Listing 41: Preconditions and postconditions of the generated `encode` method.

```
def decode(cls, codec: UPERDecoder) -> 'GeneratedClass':
    Requires(segments_valid(remaining segments))
    # ...
    Ensures(codec.segments_read_index ==
            Old(codec.segments_read_index) +
            segments_count(Old(remaining segments)))
    Ensures(Acc(Result().class_predicate()))
    Ensures(Result().is_constraint_valid_pure())
    Ensures(segments_of(Result()) ==
        codec.segments.take(segments_count(Old(remaining segments))))
```

Listing 42: Pre- and postconditions of the generated `decode` method. *Remaining segments* denotes the suffix of the codec's segment sequence at the start of the call.

While the structural role of these auxiliaries is identical across all generated classes, their definitions are necessarily type-specific. In particular, `segments_of` must capture exactly what the encoding scheme for each ASN.1 type writes to the `BitStream`, and it is through this definition that the per-type proof obligations arise.

4.2 Type-Specific Verification

Primitive ASN.1 types each encode as a single segment — for `INTEGER`, this holds for the constrained variants verified here, which are restricted to ranges of at most 64 bits — making the definitions of their auxiliary functions relatively straightforward. For `BOOLEAN`, the generated class is a frozen dataclass with a single `bool` field; since frozen instances carry no mutable state, `class_predicate` reduces to `True`. `INTEGER` types are generated as subclasses of `int`, which require no heap permissions, so `class_predicate` is again `True`. `ENUMERATED` types introduce additional structure due to the `IntEnum` wrapper generated by ASN1SCC, which is described in the following subsection. The remaining subsections address the types whose verification is more involved.

4.2.1 ENUMERATED

ASN1SCC represents an `ENUMERATED` type using two generated classes: an `IntEnum` subclass that defines the enumeration constants, and a wrapper dataclass whose single field `val` holds a value of that enum type. Listings 43 and 44 exemplify this structure for a three-valued `Status` type. During encoding, the wrapper dispatches on the enum constant and calls `encode_constrained_whole_number` with the corresponding integer index, encoding the value in $\lceil \log_2 n \rceil$ bits for n enumeration constants.

```
Status ::= ENUMERATED {
    idle(0),
    active(1),
    error(2)
}
```

Listing 43: ASN.1 definition of the example `ENUMERATED` type.

The auxiliary functions are defined in terms of the single encoded segment. The `class_predicate` asserts access to the `val` field: since `val` holds a heap-allocated `IntEnum` object (see section 5.4), ownership must be tracked explicitly. The function `segments_of` returns a one-element sequence whose segment has length two — the number of bits required for three constants — and whose value equals the integer representation of the enum constant.

```
class Status_Enum(IntEnum):
    idle = 0
    active = 1
    error = 2

@dataclass
class Status:
    val: Status_Enum = Status_Enum.idle

    def encode(self, codec: UPEREncoder) -> None:
        # ...
        if self.val == Status_Enum.idle:
            codec.encode_constrained_whole_number(0, 0, 2)
        elif self.val == Status_Enum.active:
            codec.encode_constrained_whole_number(1, 0, 2)
        elif self.val == Status_Enum.error:
            codec.encode_constrained_whole_number(2, 0, 2)
        # ...
```

Listing 44: Generated `IntEnum` and wrapper dataclass for `Status`, illustrating the encoding dispatch based on the `val` field.

The function `segments_valid` checks that exactly one segment of the correct length and with an allowed value is present. Listing 45 presents the bodies of the auxiliary definitions.

```
    def class_predicate(self) -> bool:
        return Acc(self.val)

    def is_constraint_valid_pure(self) -> bool:
        return ((self.val == Status_Enum.idle) or
                (self.val == Status_Enum.active) or
                (self.val == Status_Enum.error))

    def segments_valid(segments: PSeq[Segment]) -> bool:
        return (len(segments) > 0 and segments[0].length == 2 and
                segments[0].value in PSet(0, 1, 2))

    def segments_count(segments: PSeq[Segment]) -> int:
        return 1

    def segments_of(val: 'Status') -> PSeq[Segment]:
        return PSeq(Segment(2, int(val.val)))

    def segments_eq_lemma(left: 'Status', right: 'Status') -> bool:
        return left.val == right.val
```

Listing 45: Auxiliary functions for the `Status` wrapper class. Decorator annotations, preconditions, and postconditions are omitted for brevity.

Injectivity holds because each enum constant maps to a distinct integer value. If two `Status` instances produce the same segment, their `val` fields must carry the same integer index and therefore correspond to the same enum constant. Consequently, the two values are equal.

4.2.2 OCTET STRING

The `OCTET STRING` wrapper contains a field `arr: List[int]`, a mutable list whose elements are themselves heap-allocated. The `class_predicate` must therefore include both field access and permissions to the list contents via `list_pred`, as shown in Listing 46.

```
@Predicate
def class_predicate(self) -> bool:
    return Acc(self.arr) and Acc(list_pred(self.arr))
```

Listing 46: Generated `class_predicate` for `OCTET STRING`.

The constraint check (Listing 47) unfolds the predicate to access the list and verifies that the array has the expected length and that every element lies in the valid byte range $[0, 255]$.

```
@Pure
def is_constraint_valid_pure(self) -> bool:
    Requires(Rd(self.class_predicate()))
    Unfold(Rd(self.class_predicate()))
    return (length constraint and
            Forall(self.arr, lambda el: 0 <= el and el < 256))
```

Listing 47: Generated `is_constraint_valid_pure` for `OCTET STRING`.

The encoding produces one segment per byte, and `segments_of` (Listing 48) captures this by converting the list to its pure counterpart using `ToByteSeq` and then applying `segments_from_byteseq_full`, as described in section 3.2.

```
@Pure
@staticmethod
def segments_of(val: 'OctetString') -> PSeq[Segment]:
    Requires(Rd(val.class_predicate()) and
             val.is_constraint_valid_pure())
    Ensures(Forall(Result(), lambda seg: segment_invariant(seg)))
    Ensures(OctetString.segments_valid(Result()))
    Unfold(Rd(val.class_predicate()))
    byteseq = ToByteSeq(val.arr)
    segments = segments_from_byteseq_full(byteseq)
    return segments
```

Listing 48: Generated `segments_of` for `OCTET STRING`.

Accordingly, `segments_valid` checks that a sufficient number of segments is available and that each has length 8. The `encode` and `decode` methods delegate to the corresponding runtime library methods presented in section 3.2, whose postconditions establish and consume exactly this segment sequence.

4.2.3 SEQUENCE

The auxiliary functions for `SEQUENCE` are compositional, reflecting the sequential structure of the encoding. Listing 49 shows the example type used throughout this section.

```
MySequence ::= SEQUENCE {
    flag   BOOLEAN,
    status ENUMERATED { idle(0), active(1), error(2) }
}
```

Listing 49: ASN.1 definition of the example `SEQUENCE` type.

The `class_predicate` recursively bundles field ownership. It asserts access to each field and, for fields whose type is itself a generated class, includes the child type's own `class_predicate`, as demonstrated in Listing 50.

```
@Predicate
def class_predicate(self) -> bool:
    return (Acc(self.flag) and
            Acc(self.status) and
            self.status.class_predicate())
```

Listing 50: Generated `class_predicate` for `MySequence`.

For each field in order, `segments_of` concatenates the corresponding segment sequences. Primitive fields contribute a single segment directly, while fields of a named type delegate to that type's `segments_of`. Listing 51 shows the generated definition: the `flag` field contributes a 1-bit segment carrying the boolean value, and `status` delegates to `MySequence_status.segments_of`.

```
@Pure
@staticmethod
def segments_of(val: 'MySequence') -> PSeq[Segment]:
    Requires(Rd(val.class_predicate()) and
             val.is_constraint_valid_pure())
    # ...
    segments: PSeq[Segment] = PSeq()
    segments = segments + PSeq(Segment(1, val.flag))
    segments = segments + MySequence_status.segments_of(val.status)
    return segments
```

Listing 51: Generated `segments_of` for `MySequence`.

The function segments_count is defined accordingly as the sum of the segment counts of the individual fields, advancing through the suffix field by field, as shown in Listing 52.

```
@Pure
@staticmethod
def segments_count(segments: PSeq[Segment]) -> int:
    # ...
    count = 0
    # flag: 1 segment
    count = count + 1
    segments = segments_drop(segments, 1)
    count_status = MySequence_status.segments_count(segments)
    # status: delegate
    count = count + count_status
    segments = segments_drop(segments, count_status)
    return count
```

Listing 52: Generated segments_count for MySequence.

Validity checking follows the same pattern. The function segments_valid (Listing 53) checks each primitive field inline and delegates to the child type's segments_valid for fields of a named type. The final segments_drop after the last field has no effect on the return value. It arises from the uniform code generation strategy, in which each field is followed by an unconditional drop regardless of whether additional fields follow.

```
@Pure
@staticmethod
def segments_valid(segments: PSeq[Segment]) -> bool:
    # ...
    is_valid = True
    is_valid = (is_valid and len(segments) > 0 and
                segments[0].length == 1)
    segments = segments_drop(segments, 1)
    is_valid = is_valid and MySequence_status.segments_valid(segments)
    if is_valid:
        segments = segments_drop(segments,
                        MySequence_status.segments_count(segments))
    return is_valid
```

Listing 53: Generated segments_valid for MySequence.

The compositional structure also determines how segments_eq_lemma is established for sequences. If two sequence values produce the same concatenated segment sequence, the segments corresponding to each field position must agree. For primitive fields, equality of the field values follows directly

from equality of their segments. For fields of a named type, the corresponding child `segments_eq_lemma` is invoked to conclude equality of those fields. Applying this argument field by field yields equality of the two sequence values.

4.3 Invertibility

The central property established by the verification is round-trip correctness: for any value that satisfies the schema constraints, decoding the result of encoding that value recovers the original. Figure 4.1 summarises the argument.

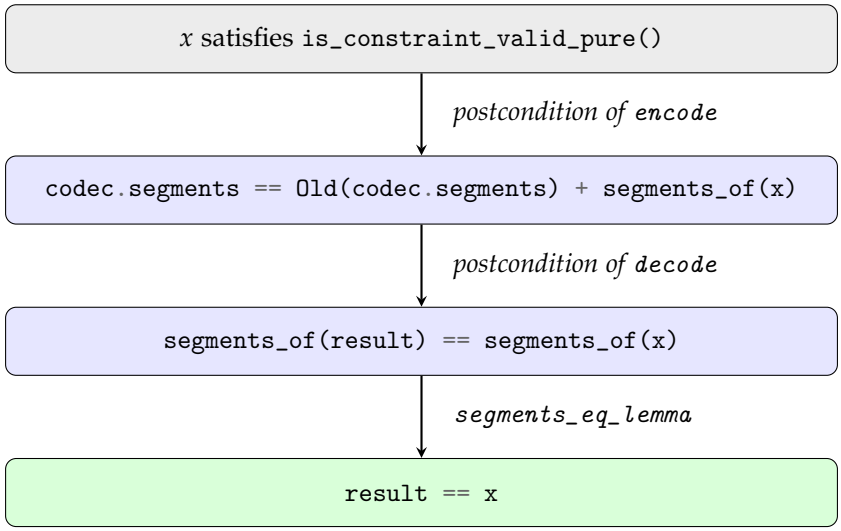

Figure 4.1: The three-step round-trip argument for any verified ASN.1 type.

First, the postcondition of `encode` establishes that the codec's segment sequence has been extended by `segments_of(x)`. Second, the postcondition of `decode` guarantees that the segment sequence of the decoded result equals the corresponding prefix of the codec's segments, which by construction is `segments_of(x)`. Third, `segments_eq_lemma` concludes from this equality that the decoded result is identical to `x`.

The design of the auxiliary functions ensures that these three steps compose without requiring any type-specific reasoning at the top level. The argument operates entirely at the segment level, without reference to the underlying bit representation — precisely the guarantee provided by the segment abstraction of the runtime library. As established in Chapter 3, the `segments_contained` invariant of the `BitStream` ensures that each segment faithfully captures the bits written to the buffer during encoding and read back during decoding. Segment-level equality is therefore sufficient to conclude full correctness.

For BOOLEAN, INTEGER, and ENUMERATED, the argument requires no further elaboration, as segments_of is defined directly in terms of a single segment. For SEQUENCE, the same three steps apply but unfold compositionally. The MySequence type from Listing 49 illustrates this compositional structure.

```
@Pure
@Opaque
@staticmethod
def segments_eq_lemma(
        left: 'MySequence',
        right: 'MySequence') -> bool:
    Requires(Rd(left.class_predicate()) and
             left.is_constraint_valid_pure())
    Requires(Rd(right.class_predicate()) and
             right.is_constraint_valid_pure())
    Requires(MySequence.segments_of(left) ==
            MySequence.segments_of(right))
    Ensures(left == right)
    # ...
    Assert(segments_left[0] == segments_right[0])        # flag
    # ...
    Assert(MySequence_status.segments_eq_lemma(left.status,
                                               right.status))
    return True
```

Listing 54: Generated segments_eq_lemma for MySequence.

Listing 54 shows the generated segments_eq_lemma for MySequence. The postcondition of MySequence.encode establishes that the codec contains the concatenation of the segment for flag and the segment sequence for status. After decoding, the postcondition of MySequence.decode establishes that segments_of(result) equals that same concatenation. Since the concatenated sequences are equal, the segments corresponding to each field must also be equal.

For flag, equality of the decoded value follows directly from equality of its segment. For status, the child segments_eq_lemma is invoked to conclude that the decoded status value equals the original. Together, these field-level equalities imply equality of the two MySequence values, thereby completing the round-trip proof.

4.4 Error Handling

The specifications in this chapter assume well-formed input: encoding requires is_constraint_valid_pure, and decoding requires segments_valid. When these preconditions are violated, no formal guarantees are provided.

In such cases, two categories of errors may arise: buffer errors and semantic errors.

The first category, buffer errors, concerns accesses that exceed the buffer boundary. For encoding, this situation is prevented by the following precondition: `Requires(codec.remaining_bits >= required bits)`. If it is violated, a `BitStreamError` is raised by the underlying `BitStream`. For decoding, `segments_valid` serves an analogous role: without this condition, the decoder may attempt to read beyond the buffer.

The second category, semantic errors, concerns the validity of the decoded result. Without the assurance provided by `segments_valid`, the decoder may produce a value that violates the ASN.1 constraints, and therefore the guarantee `Result().is_constraint_valid_pure()` can no longer be established. For dynamically sized types, the two categories may interact: a malformed length field may cause the decoder to read more data than the buffer contains, effectively turning a semantic error into a buffer error.

The runtime implementation handles both categories. As described in section 3.3, `BitStream` raises `BitStreamError` on buffer violations (Listing 36). The `Encoder` and `Decoder` methods catch these exceptions and wrap them into `EncodeResult` and `DecodeResult` objects (Listing 37). The generated classes then inspect these results and propagate failures as `Asn1Exception`.

Listing 55 illustrates this pattern for the generated `encode` method of `BOOLEAN`: the method first validates the input against the ASN.1 constraints and then checks the codec result to guard against buffer errors. Listing 56 demonstrates the symmetric pattern for `decode`: the `DecodeResult` is checked for buffer errors, and the decoded instance is validated against the constraints to catch semantic errors.

```
def encode(self, codec: UPEREncoder) -> None:
    # ...
    valid_res = self.is_constraint_valid()
    if not valid_res.is_valid:
        raise Asn1Exception(...)
    # ...
    res = codec.append_bit(bool(self))
    if not res.success:
        raise Asn1Exception(...)
    # ...
```

Listing 55: Error handling in the generated `encode` method for `BOOLEAN`: constraint validation followed by a buffer error check.

This error-handling infrastructure is implemented in the runtime system but is not formally specified. The current verification mandates sufficient

capacity and valid input data through its preconditions, so the verifier never reasons about the failure paths.

```
@classmethod
def decode(cls, codec: UPERDecoder) -> 'BasicBool':
    # ...
    decoded_BasicBool = codec.read_bit()
    if (not decoded_BasicBool.success or
            decoded_BasicBool.decoded_value is None):
        raise Asn1Exception(...)
    instance = BasicBool(decoded_BasicBool.decoded_value)
    # ...
    valid_res = instance.is_constraint_valid()
    if not valid_res.is_valid:
        raise Asn1Exception(...)
    # ...
```

Listing 56: Error handling in the generated decode method for BOOLEAN: buffer error checking followed by constraint validation of the decoded value.

Extending the verification to cover these cases would require replacing preconditions with conditional postconditions of the form:

```
Ensures(Implies(precondition, postcondition))
```

Exceptional outcomes could additionally be specified using

```
Exsures(ExceptionType, state)
```

which characterises the heap state when a particular exception is raised.

If applied consistently throughout the runtime library and the generated classes, this approach would allow the decoder to accept arbitrary input and either produce a verified result or raise a verified exception. However, achieving this would require substantial effort, as the resulting conditional structure would need to be propagated through every method in the call chain.

Chapter 5

Nagini Extensions

This chapter documents the contributions made to Nagini in the course of this project. The goal is not to provide full general-purpose support for the introduced features. Rather, each extension is tailored to the requirements of verifying the ASN.1 encoder and decoder pipeline.

5.1 Bytearray

The `BitStream` stores binary data in a Python `bytearray`, a mutable sequence of bytes. To support this, we extended Nagini with a model of `bytearray`, representing each instance in Viper as a `Seq[Int]`, following the same representation used for Python lists.

Access to the underlying sequence is governed by a `bytearray_pred` predicate, which must be held prior to any read or write operation. This design integrates naturally with Viper's permission-based verification model — full permission is required for all methods modifying the state, while any fractional permission amount is sufficient for read-only functions. The extension supports construction from an integer length, from a list of integers, and from another `bytearray`. The complete set of supported operations is summarised in Table 5.1.

Preconditions and postconditions on all operations ensure that every stored value lies within the byte range $[0, 255]$, which is essential for maintaining the invariants of the `BitStream`. Because `bytearray` is a mutable type, its data resides on the heap and therefore requires access permissions. Consequently, all methods that modify its state are impure and cannot be used in specifications or pure functions.

Function	Description
`__init__(...)`	Construct new `bytearray`
`__getitem__(index)`	Indexed byte read
`__getitem_slice__(i, j)`	Slice read returning a `bytearray`
`__setitem__(index, val)`	Indexed byte write
`__len__()`	Number of bytes
`__contains__(val)`	Membership test
`__iter__()`	Iterator over bytes
`__bool__()`	`False` iff the sequence is empty
`__eq__(other)`	Comparison with another `bytearray`
`append(val)`	Append a single byte
`extend(other)`	Append all bytes of another `bytearray`
`reverse()`	Reverse the sequence in place

Table 5.1: Supported interface of `bytearray`.

5.1.1 Pure Byte Sequences

To enable the use of byte sequences in specifications and pure functions, we introduced `PByteSeq`, a pure, immutable sequence type also backed by `Seq[Int]`. As a mathematical data type similar to `Sequence` or `PSet`, `PByteSeq` can be freely constructed and manipulated within specifications.

Instances of `PByteSeq` enforce the same $[0, 255]$ bounds on their elements as `bytearray`. The function `ToByteSeq` converts a `bytearray` or a `list[int]` into a `PByteSeq`. In the latter case, the byte bounds are discharged as proof obligations. The interface for `PByteSeq` mirrors that of the existing `Sequence` type and is summarised in Table 5.2.

Function	Description
`__init__(...)`	Construct new `PByteSeq`
`__getitem__(index)`	Indexed byte access
`__len__()`	Number of bytes
`__contains__(item)`	Membership test
`__iter__()`	Iterator over bytes
`__add__(other)`	Concatenation of two sequences
`take(n)`	Prefix of the first n bytes
`drop(n)`	Suffix after the first n bytes
`update(i, v)`	Sequence with byte i replaced by v
`ToByteSeq(l)`	Convert l to `PByteSeq`

Table 5.2: Interface of `PByteSeq`.

5.2 Bit Operations

The `BitStream` implementation relies on the left-shift (`<<`) and right-shift (`>>`) operators to manipulate bits within bytes, as well as on `bit_length` to determine the minimum number of bits required to represent an integer. Prior to this project, none of these operations were supported by Nagini.

A direct encoding of shift operations using SMT-LIB bitvector primitives proved impractical due to prohibitive verification times (see section 3.1.5). Instead, shift operations are encoded using integer arithmetic. In particular, a left shift by k is equivalent to multiplication by 2^k. The factor 2^k is resolved through a case distinction over the shift amount, resulting in a family of Viper helper functions that collectively cover shifts of up to 64 bits, as illustrated in Listing 57. For shift amounts exceeding 64 bits, the implementation falls back to SMT-LIB bitvector operations.

```
function __prim__int___lshift__(self: Int, other: Int): Int
    requires ...
{
    0 <= other <= 64 ? self * __shift_factor64(other) :
    0 <= self ?
            fromBVInt(shlBVInt(toBVInt(self), toBVInt(other))) :
            -fromBVInt(shlBVInt(toBVInt(-self), toBVInt(other)))
}

function __shift_factor64(amount: Int): Int
    requires 0 <= amount && amount <= 64
{
    amount <= 32 ? __shift_factor32(amount) :
            __shift_factor32(amount - 32) * __shift_factor32(32)
}

function __shift_factor32(amount: Int): Int
    requires 0 <= amount && amount <= 32
{
    amount <= 8 ?  __shift_factor8(amount) :
    // ...
}

function __shift_factor8(amount: Int): Int
    requires 0 <= amount && amount <= 8
{
    amount == 0 ? 1 :
    amount == 1 ? 2 :
    // ...
}
```

Listing 57: Integer-arithmetic Viper encoding of left shift with fallback.

The `bit_length` method is handled analogously through a case distinction over value ranges. Specifically, the definition $2^{k-1} \leq |x| < 2^k$ is used to determine the corresponding value of k for each interval. Since the method operates on the magnitude of the input, the encoding first takes the absolute value before delegating to a hierarchy of Viper helper functions, each responsible for a progressively smaller range, as illustrated in Listing 58

```
function __prim__int_bit_length(val: Int): Int
    requires 0 <= val && val < __shift_factor64(64)
{
    __int_bit_length64(abs(val))
}

function __int_bit_length64(val: Int): Int
    requires 0 <= val && val < __shift_factor64(64)
{
    val < __shift_factor32(32) ? __int_bit_length32(val) :
          __int_bit_length32(val / __shift_factor32(32)) + 32
}

function __int_bit_length32(val: Int): Int
    requires 0 <= val && val < __shift_factor32(32)
{
    val < __shift_factor32(8) ? __int_bit_length8(val) :
    val < __shift_factor32(16) ?
            __int_bit_length8(val / __shift_factor32(8)) + 8  :
    // ...
}

function __int_bit_length8(val: Int): Int
    requires 0 <= val && val < 256
{
    val < 1 ? 0 :
    // ...
    val < 128 ? 7 : 8
}
```

Listing 58: Viper encoding of `bit_length` via case distinction on value bounds.

5.3 Dataclass

Python dataclasses provide a concise syntax for defining classes that primarily serve as data containers. We extended Nagini with support for the `@dataclass` decorator, covering both frozen (immutable) and non-frozen dataclasses, and supporting the forms used by the ASN1SCC Python backend.

```python
@dataclass
class MyClass:
    boolField: bool
```

Listing 59: A simple Python dataclass definition.

Dataclasses declare fields within the class body and rely on an implicitly generated `__init__` method. During Nagini's analysis phase, this method is synthesised and inserted into the Python AST, after which it is translated in the same manner as any user-defined method.

The translation differs depending on whether the dataclass is frozen. Non-frozen dataclasses are translated analogously to regular classes (Listing 60): fields are represented as heap-allocated attributes and are assigned within the `__init__` method. In contrast, for frozen dataclasses, each field is translated into a pure function that returns the value passed to the corresponding parameter of `__init__` (Listing 61).

```
field MyClass_boolField: Ref

method MyClass___init__(self: Ref, boolField: Ref)
  requires self != null
  requires issubtype(typeof(self), MyClass())
  requires issubtype(typeof(boolField), bool())
  ensures acc(self.MyClass_boolField, write) &&
    issubtype(typeof(self.MyClass_boolField), bool())
  ensures self.MyClass_boolField == boolField
```

Listing 60: Generated Viper encoding for `MyClass`.

```
function MyClass_boolField_getter(self_0: Ref): Ref
    requires issubtype(typeof(self_0), MyClass())
    requires self_0 != null
    ensures issubtype(typeof(result), bool())
    decreases _

method MyClass___init__(self: Ref, boolField: Ref)
    requires self != null
    requires issubtype(typeof(self), MyClass())
    requires issubtype(typeof(boolField), bool())
    ensures MyClass_boolField_getter(self) == boolField
```

Listing 61: Generated Viper encoding for `MyClass` with `dataclass(frozen=True)`.

Default Values

Default values for dataclass fields are supported in two forms. Simple literal defaults, such as `x: int = 0`, are translated into default arguments of the synthesised `__init__`, as shown in Listing 62. Since Nagini already supports default arguments for regular methods, no additional handling is required in this case.

```
@dataclass
class MyClass:
    x: int = 0
```

```
def __init__(self, x: int = 0):
    Ensures(Acc(self.x) and self.x == x)
```

Listing 62: Dataclass with a literal default value (top) and the synthesised `__init__` method (bottom).

Factory-based defaults, specified via `field(default_factory=list)`, require different treatment, as the factory must be invoked separately for each instance. To achieve this, the synthesised `__init__` method uses `None` as a sentinel default argument, and Nagini inserts an invocation of the default factory method at the call site when no value is provided by the caller. In the generated Viper code (Listing 63), this corresponds to first constructing a new list instance and subsequently passing it to `__init__`.

```
@dataclass
class MyClass:
    arr: List[int] = field(default_factory=list)
```

```
list_0 := list___init__()
inhale issubtype(typeof(list_0), list(int()))
MyClass_res := new()
inhale typeof(MyClass_res) == MyClass()
MyClass___init__(MyClass_res, list_0)
```

Listing 63: Python dataclass with a factory default (top) and the corresponding Viper encoding of a constructor call (bottom).

Post Init

An additional feature of dataclasses is the optional `__post_init__` method, which is automatically invoked by Python after `__init__` completes and may define its own pre- and postconditions. When such a method is present and no custom `__init__` is defined, each construction site is augmented with an explicit call to `__post_init__` immediately following `__init__`, thereby

preserving the intended execution order. This is illustrated in Listing 64: the Python dataclass defines a `__post_init__` method that increments `x` after construction. The generated Viper code reflects this by invoking `MyClass___post_init__` directly after `MyClass___init__` at the construction site.

```python
@dataclass
class MyClass:
    x: int

    def __post_init__(self) -> None:
        Requires(Acc(self.x))
        Ensures(Acc(self.x))
        Ensures(self.x == Old(self.x) + 1)

        self.x += 1

m = MyClass(5)
```

```
method main()
{
    // ...
    MyClass_res := new()
    inhale typeof(MyClass_res) == MyClass()
    MyClass___init__(MyClass_res, __prim__int___box__(5))
    MyClass___post_init__(MyClass_res)
    // ...
}

method MyClass___post_init__(self: Ref)
    // ...
```

Listing 64: Dataclass with a `__post_init__` method carrying a Nagini specification.

5.4 IntEnum

Python's `IntEnum` defines an enumerated type that inherits from `int`, thereby permitting numeric comparisons and arithmetic operations. In Nagini, each `IntEnum` is encoded using a pair of boxing and unboxing functions that mediate between raw integers and typed enum references, while enforcing constraints on the set of admissible values.

For each named constant, a dedicated function is generated that directly returns the corresponding boxed value. Listing 65 illustrates this encoding for a two-valued flag type.

```
class Flag(IntEnum):
    success = 0
    failure = 1
```

```
function Flag__int__(box: Ref): Int
    requires typeof(box) == Flag()
    ensures false || result == 0 || result == 1
    ensures (forall i: Ref ::
        { object___eq__(box, i), Flag__int__(i) }
        object___eq__(box, i) && typeof(i) == Flag() ==>
        Flag__int__(i) == result)
    decreases _

function Flag__box__(value: Int): Ref
    requires false || value == 0 || value == 1
    ensures typeof(result) == Flag()
    ensures Flag__int__(result) == value
    ensures int___unbox__(result) == value
    decreases _

function Flag_success(receiver: PyType): Ref
    ensures typeof(result) == Flag()
    ensures receiver == Flag() ==> result == Flag__box__(0)

function Flag_failure(receiver: PyType): Ref
    ensures typeof(result) == Flag()
    ensures receiver == Flag() ==> result == Flag__box__(1)
```

Listing 65: Python `IntEnum` definition (top) and the corresponding Viper encoding (bottom).

The pattern `false || result == 0 || result == 1` in the postcondition is an artifact of code generation. The generator initializes the condition with `false` and extends the condition with each permitted value via logical disjunction.

At use sites, while attribute accesses such as `Flag.success` are translated into calls to `Flag_success`, constructor calls such as `Flag(0)` are converted into `Flag__box__(0)`. When a value of type `Flag` appears in a numeric comparison, Nagini inserts a call to `Flag__int__` to extract the underlying integer. This enables comparisons both with plain integers and with values of other `IntEnum` types, as illustrated in Listing 66.

```
a = flag.failure
b = a + 5

Assert(b == 6)
```

Listing 66: Objects of `IntEnum` behave as integers.

5.5 Additional Features

Beyond the core language features described above, several smaller extensions were implemented to improve the usability of Nagini in the context of this project.

To avoid introducing a runtime dependency on Nagini in production code, we added a preprocessing mode. In this mode, Nagini annotations are written as Python comments prefixed with `@nagini`, ensuring the annotated file remains valid Python code that can be executed independently of Nagini. The preprocessor strips the comment prefix before passing the file to Nagini for verification, as demonstrated in Listing 67.

```
def foo(x: int) -> int:
    #@nagini Requires(0 <= x)
    # ...
```

```
def foo(x: int) -> int:
    Requires(0 <= x)
    # ...
```

Listing 67: A function annotated with the comment prefix (top) and the preprocessed form passed to Nagini for verification (bottom).

The current implementation is limited to single files; annotations in imported modules are not processed. Furthermore, a function-level `PInt` annotation (see section 6.2) is required to use `PInt` in conjunction with the preprocessing mode, since parameter-level annotations on imported symbols are not handled.

We also added support for Python's f-string syntax. While the implementation does not provide guarantees about the resulting string values, it ensures that all expressions embedded within a format string satisfy their preconditions. This enables verification to proceed through code that uses format strings for logging or error reporting, without requiring full reasoning about string contents.

Additionally, we introduced a `--benchmark-timeout` flag to Nagini, which allows verification runs to be terminated after a configurable duration. For example:

```
nagini test.py --benchmark --benchmark-timeout 120
```

Finally, the `PInt` annotation was added, which instructs Nagini to treat an integer parameter as a primitive, unboxed value. Its motivation and usage are discussed in Chapter 6.

5.6 Reported Bugs

During the course of the project, several bugs were identified in Nagini and reported to the issue tracker.[1]

Six crashes were reported, each accompanied by a minimal reproducing test case. All of these were subsequently fixed by the Nagini team, and the provided tests were incorporated into the repository to prevent regressions.

We also identified and reported three soundness issues, all of which were later resolved. One issue arose from Python's ability to define integer subclasses: because a subclass instance satisfies `A(5) == 5`, Nagini was misled into accepting the trivially false assertion that `2 == 1`, as illustrated in Listing 68. A second issue affected object casting, and a third affected string equality.

```
class A(int):
    pass

assert A(5) == 5
assert 2 == 1
```

Listing 68: Unsound behavior for integer subclasses: the false assertion on the last line was accepted by Nagini.

Relative imports of the form `from .module import Symbol` were initially unsupported and caused errors. This issue was reported and subsequently fixed. In addition, we reported several additional issues related to completeness, type handling, and the use of generic types, each with a minimal reproducing test case. All of these issues were eventually resolved.

[1] https://github.com/marcoeilers/nagini/issues

Chapter 6

Proof Structure and Challenges

This chapter describes the proof-structuring techniques employed and the challenges encountered during the verification work. The first section presents the techniques used throughout the project: leveraging the opacity mechanism to isolate function contracts, decomposing proofs into focused helper lemmata to enhance manageability, and splitting case distinctions into separate lemmata to reduce verification time. The second section discusses a challenge arising from Python's handling of integer types and the annotation introduced to address it in verification.

6.1 Opacity, Lemma Extraction, and Case Splitting

Throughout the verification, functions are declared as both pure and opaque. Marking a function as opaque hides its body from callers: when the function is invoked, the verifier can only reason about its behavior using the specified preconditions and postconditions and cannot inspect the implementation. This separation is intentional: postconditions define precisely what the function exposes to the rest of the proof, while the verification of the function body is handled independently. If a proof requires inspecting the body of a pure opaque function, this is done explicitly via `Reveal()`, which unfolds the function definition at that specific point.

Lemmata follow the same pure-opaque structure but serve a distinct purpose: the body of each lemma constitutes a proof that establishes the postcondition, and is never revealed at call sites. Invoking a lemma makes its postcondition available as a verified fact without re-executing the proof. Proofs are then assembled by invoking only the lemmata whose conclusions are required at each step.

A key structural decision is determining where to place a property: as a postcondition of the function that computes the value, or as a separate

lemma about that function. For example, `byteseq_set_bits` (described in section 3.1.2) writes a sequence of bits into a byte sequence and returns the result. All its properties could be placed directly as postconditions, as shown in the top half of Listing 69. However, the read-back and prefix-preservation properties are extracted into the separate lemma `lemma_byteseq_set_bits`, shown in the bottom half.

```
# Without extraction: all properties on the function itself
@Pure
@Opaque
def byteseq_set_bits(args) -> PByteSeq:
    Ensures(len(seq) == len(Result()))
    Ensures(byteseq_equal_until(Result(), seq, position))
    Ensures(byteseq_read_bits(Result(), position, length) == value)
    # ...
```

```
# With extraction: structural postcondition only
# properties in a separate lemma
@Pure
@Opaque
def byteseq_set_bits(args) -> PByteSeq:
    Ensures(len(seq) == len(Result()))
    # ...

@Pure
@Opaque
def lemma_byteseq_set_bits(args) -> bool:
    Ensures(byteseq_equal_until(
        byteseq_set_bits(args), byteseq, position))
    Ensures(byteseq_read_bits(
        byteseq_set_bits(args), position, length) == value)
    # ...
```

Listing 69: Lemma extraction applied to `byteseq_set_bits`. The read-back and prefix-preservation properties are moved from the function's postconditions into a dedicated lemma.

This separation offers two key advantages. First, any function that calls `byteseq_set_bits` but does not require the read-back or prefix-preservation properties incurs no additional verification cost — the corresponding lemma simply remains uninvoked. Second, proofs that do require these properties can invoke `lemma_byteseq_set_bits` once, obtaining them as verified facts without needing to re-establish them at each use site. This pattern generalizes to any scenario where a function's properties are only relevant in specific proof contexts.

Complex proofs are often decomposed into multiple such helper lemmata, even when a single lemma could, in principle, establish all properties. The primary motivation is manageability: dividing a proof into focused steps

simplifies construction and maintenance, and ensures that individual proof obligations remain small enough to reason about directly. In practice, this decomposition introduces a minor verification overhead compared to proving all properties in a single lemma, as each additional function boundary imposes a small amount of extra work for the verifier. However, the difference is generally negligible and is more than compensated by the clarity and maintainability gained, as illustrated by the `lemma_byteseq_set_bits` benchmark in Figure 6.1.

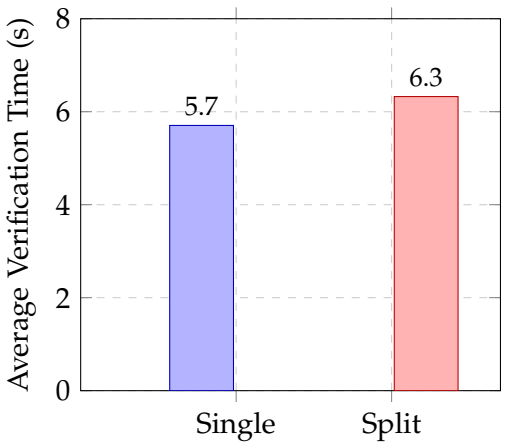

Figure 6.1: Average verification time for the byteseq-set-bits test case (Listing 76 on page 96) comparing a single merged lemma with two separate lemmata.

An exception arises when a lemma involves a case distinction over a concrete value, such as a boolean argument. In such situations, splitting the proof into one helper lemma per case — rather than handling both in a single lemma — allows the SMT solver to reason about each branch in isolation, with the case condition fixed. This approach can lead to a significant reduction in verification time.

A concrete example involves proving that `byte_set_bit`, which sets a single bit in a byte, and `byte_set_bits`, which writes a field of consecutive bits, agree when the length argument is 1 — a key lemma in the verification of the runtime library. Establishing this equivalence requires reasoning about both the `True` and `False` branches of `byte_set_bit`'s bit argument. By splitting the proof into two helper lemmata — one per branch — verification time is reduced by approximately one third compared to the single-lemma approach, as shown in Figure 6.2.

In both strategies — extracting properties into separate lemmata and splitting case distinctions — the underlying motivation is the same: decomposing a proof into smaller, focused obligations keeps it tractable both for the human constructing it and for the SMT solver discharging it.

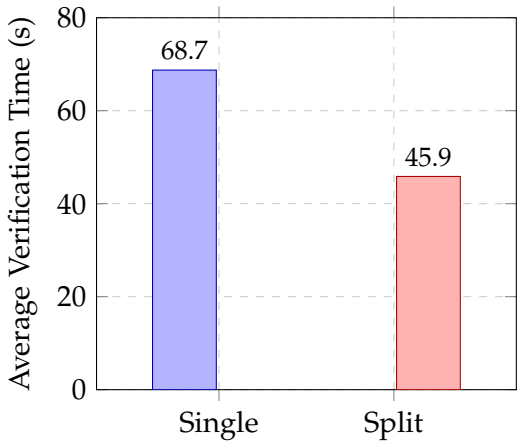

Figure 6.2: Average verification time for the set-equal test case (Listing 74 on page 90) using a single combined lemma versus two case-split lemmata.

6.2 Python Integer Boxing

A separate challenge arises not from proof structure but from how Nagini models Python integers. Python's type system permits subclassing of `int`, which means that at runtime an integer value may be an instance of a subclass. To accommodate this, Nagini represents Python integers as boxed heap objects of type `Ref` rather than as primitive Viper integers. An unboxing function, `int___unbox__`, is used to extract the underlying integer value when arithmetic reasoning is required.

The necessity of this approach is illustrated by the following scenario: even if the condition `x == y` holds, the assertion `f(x) == f(y)` is not guaranteed because `f` may use `isinstance` to distinguish `bool` from `int` — a distinction that would be lost if integers were represented as primitive values.

```
def bar(x: int, y: int):
    Requires(x == y)

    f_x = f(x)
    f_y = f(y)
    Assert(f_x == f_y)  # cannot be verified

def f(a: int) -> int:
    if isinstance(a, bool):
        return 0
    else:
        return a
```

Listing 70: Two equal integers may behave differently if one is an instance of a subclass.

In most contexts, this boxing is transparent. However, it has a significant consequence when reasoning about pure opaque functions. Since the verifier tracks both the type and the unboxed value of each boxed integer, two parameters with equal numeric values are not considered interchangeable at an opaque call site as their runtime types may differ. Consequently, `x == y` alone is insufficient to conclude `f(x) == f(y)` when `f` is opaque — the function body would need to be revealed at each call site to determine whether the result depends on type information. Alternatively, conditions such as `x is y` or the combination `x == y and type(x) is int and type(y) is int` would suffice, as they both establish that the two arguments are indistinguishable to the verifier. However, adding such conditions at every call site is impractical.

To address this, the `PInt` annotation was introduced in Nagini during the course of this project (see Chapter 5). Annotating a parameter with `PInt` instructs Nagini to represent it as a primitive, unboxed Viper `Int`, discarding the subclass information. With `PInt`, two parameters with equal values are truly indistinguishable to the verifier, so `x == y` implies `f(x) == f(y)` without revealing the body of `f`, thus preserving the benefits of the opaque pattern.

Throughout the project, `PInt` is applied to the parameters of low-level bit-operation functions such as `byte_set_bit` and `byte_read_bit`. These functions are pure and opaque, and are invoked at many call sites with arguments that are known to be equal by value. Without `PInt`, the verifier cannot conclude that two equal integer arguments produce equal results from an opaque function, requiring either the body to be revealed at each call site or additional lemmata to establish equality. Annotating the parameters with `PInt` removes this friction, allowing the opaque contract to be used directly wherever these functions appear.

6.2.1 Soundness

The use of `PInt` is sound provided that the annotated function does not inspect the runtime type of its parameters via `isinstance` or similar mechanisms that distinguish `int` subclasses, such as `bool`, as illustrated in Listing 70. Currently, this condition must be verified manually for every function whose parameters are annotated with `PInt`. Ideally, Nagini would enforce this constraint automatically.

In this project, all such functions perform only arithmetic and bitwise operations on their integer parameters, without any `isinstance` checks, ensuring that the annotation is applied soundly. A function-level `PInt` annotation — which would mark all integer parameters of the annotated function as primitive in a single declaration — would provide a more concise interface and make the scope of the soundness obligation explicit. Implementing this feature remains a potential future extension to Nagini.

Chapter 7

Evaluation

This chapter presents an evaluation of the verification work conducted in this thesis. We first summarise the components verified within the runtime library and the generated data classes, and report the verification effort in terms of time and annotation overhead. We then compare the approach with the prior verification work in Scala, discuss practical benefits of the verification effort beyond the formal proofs themselves, and conclude with the limitations of the current scope.

7.1 Coverage

The verification covers two components: the runtime library and the generated data classes. Verification is restricted to uPER encoding. All verified properties focus on the happy path: encoding correctness is established for constraint-satisfying inputs, and decoding assumes that a valid encoding is present in the buffer. Error handling and behavior under invalid-input conditions fall outside the scope of this work, as discussed in section 4.4.

Table 7.1 lists the codec method groups of the `Encoder` and `Decoder` classes. For each group, the corresponding read or decode method was verified alongside the encode or write method. These verified methods form a representative subset of the complete codec. Remaining uPER encoding variants, as well as all ACN methods, are not covered.

The bit/byte primitive and alignment groups constitute the foundational layer: every higher-level encoding method ultimately delegates to these operations, so their verification establishes the basis for the type-specific proofs. The integer and octet string groups cover the uPER encoding variants invoked by the ASN.1 types verified at the data class level (Table 7.2). The null group requires only advancing the bit position and is included for completeness.

Method	Verified
Bit/byte primitives	
`append_bit`	✓
`append_byte`	✓
`append_byte_array`	✓
`append_bits`	✓
Alignment	
`align_to_byte`	✓
`align_to_word`	✓
`align_to_dword`	✓
Null	
`encode_null`	✓
Integer	
`encode_integer`	✓
`encode_unsigned_integer`	✓
`encode_constrained_pos_whole_number`	✓
`encode_constrained_whole_number`	✓
Octet string	
`encode_octet_string_no_length`	✓
`encode_octet_string_no_length_vec`	✓
Remaining uPER variants	
`encode_semi_constrained_whole_number`	—
`encode_semi_constrained_pos_whole_number`	—
`encode_unconstrained_whole_number`	—
`enc_real`	—
`encode_bit_string`	—
ACN encoding	
(all methods)	—

Table 7.1: Coverage of runtime library codec methods. Each encode or write method was verified together with its corresponding decode or read counterpart.

Table 7.2 summarises the supported ASN.1 types. For each verified type, round-trip correctness was proved: encoding a constraint-satisfying value and subsequently decoding it reproduces the original value.

ASN.1 Type	Verified	Notes
BOOLEAN	✓	
ENUMERATED	✓	
NULL	✓	
INTEGER	✓	Constrained range only
OCTET STRING	✓	Fixed size only
SEQUENCE	✓	No optional fields, statically-sized children only
SEQUENCE OF	—	
CHOICE	—	
BIT STRING	—	
IA5String	—	
REAL	—	

Table 7.2: Coverage of ASN.1 types. Round-trip correctness is proved for all covered types.

7.2 Verification Effort

Table 7.3 reports the total verification time per component. Verifying all functions in a file simultaneously often overwhelms the verifier; therefore, each function was verified individually. For the runtime library, the values shown represent the sums of wall-clock times of those individual runs, while a per-function breakdown is provided in Appendix A. For the generated data classes, each entry corresponds to the total time required to verify a single class. All times were measured on the machine described in Table A.1.

Because each function is verified in isolation, the reported times include per-invocation startup overhead from the Nagini and Viper toolchain. This overhead is substantial relative to the actual proof work for simple functions, so the totals overstate the time that would be needed if all functions could be verified in a single run.

The generated data classes require significantly less time per class than the runtime library files. This reflects the design of the verification architecture: auxiliary functions in the generated classes are defined in terms of the segment abstraction, which encapsulates all bit-level reasoning behind the runtime library's postconditions. Consequently, the SMT obligations at the data class level are much lighter than those in the runtime library, where the segment abstraction itself must be established.

Component	Time (s)
Runtime library	
`asn1_types.py`	846.1
`verification.py`	1'214.3
`segment.py`	964.9
`bitstream.py`	1'042.0
`codec.py`	433.8
`decoder.py`	1'150.6
`encoder.py`	867.3
`codec_uper.py`	19.7
Generated data classes (see Listing 71)	
`BasicBool`	35.9
`MyInt`	31.1
`MyOctetString`	80.9
`Status`	40.9
`ShortSequence`	89.0
`NestedSequence`	110.6

Table 7.3: Verification times per component. Runtime library entries are sums of per-function times; generated data class entries are per-class totals.

Table 7.4 reports the annotation overhead for both the runtime library and the generated data classes. Excluding the ACN files discussed below, 1'636 of 4'097 lines (39.9 %) in the runtime library correspond to specification code. Two files — `segment.py` and `verification.py` — consist entirely of specification code, as they have no runtime counterparts and exist solely to support the proof.

Among the implementation files, `bitstream.py` exhibits the highest annotation burden (68.0 %), reflecting the effort required to establish the segment abstraction at the byte-sequence level. In contrast, `encoder.py` and `decoder.py` contain comparatively few annotations (12.6 % and 13.5 % respectively): their postconditions are expressed directly in terms of segments, and the segment abstraction isolates them from the bit-level reasoning concentrated in `bitstream.py`.

Not all methods in `encoder.py` and `decoder.py` have been verified yet; the unverified methods correspond to the remaining uPER variants listed in Table 7.1. Consequently, the annotation counts reflect only the subset of functions for which specifications have been provided. The ACN counterparts, `acn_encoder.py` and `acn_decoder.py`, carry no annotations, as ACN encoding lies entirely outside the scope of this work.

The generated data classes exhibit a higher overall specification ratio (54.1 %) than the runtime library, reflecting the uniform set of auxiliary functions introduced into each generated class. The sole exception is `Status_Enum`: as a plain `IntEnum` subclass, its verification is handled entirely by the Nagini extension described in section 5.4, without requiring any class-level annotations.

File / Class	Impl	Spec	Total	Spec %
Runtime library				
`asn1_constants.py`	21	0	21	0.0
`asn1_exceptions.py`	43	10	53	18.9
`asn1_types.py`	363	34	397	8.6
`bitstream.py`	169	359	528	68.0
`codec.py`	91	71	162	43.8
`codec_uper.py`	23	4	27	14.8
`decoder.py`	978	152	1′130	13.5
`encoder.py`	773	111	884	12.6
`acn_decoder.py`	1′336	0	1′336	0.0
`acn_encoder.py`	1′395	0	1′395	0.0
`segment.py`	0	309	309	100.0
`verification.py`	0	586	586	100.0
Total	**5′192**	**1′636**	**6′828**	**23.9**
Generated data classes (see Listing 71)				
`BasicBool`	55	76	131	58.0
`MyInt`	65	81	146	55.5
`MyOctetString`	68	108	176	61.4
`Status_Enum`	6	0	6	0.0
`Status`	96	81	177	45.8
`ShortSequence`	104	128	232	55.2
`NestedSequence`	121	134	255	52.5
Total	**515**	**608**	**1′123**	**54.1**

Table 7.4: Lines of implementation (Impl) and specification (Spec) per runtime library file and generated data class. Spec % denotes the proportion of lines dedicated to specification relative to the total number of lines.

7.3 Comparison with the Scala Verification

The verification work of Bucev et al. [5] on the Scala backend provides a natural point of comparison for this thesis. Both efforts target ASN1SCC at the same level: the runtime library and the generated data classes. Both prove round-trip correctness for selected ASN.1 data types and introduce an intermediate representation to decouple bit-level buffer reasoning from codec-level proofs. The primary differences lie in the encoding scheme, the intermediate representation, the composition of proofs, and the infrastructure investment required.

Encoding Scheme

Bucev et al. focus primarily on ACN encoding, the scheme ESA uses in practice for the PUS-C standard [1]. This thesis only targets uPER encoding, which is a prerequisite for ACN. Both schemes share the same underlying bit operations and the overall structure; the differences appear in the type-specific codec methods and the constraints they must satisfy. Extending the Python verification work to ACN would require adding library methods with ACN-specific postconditions, but the segment abstraction and round-trip proof structure remain directly applicable.

Intermediate Representation

The central challenge in both works is that bit operations can cross byte boundaries, making direct buffer reasoning intractable at the codec level. Bucev et al. address this using a bit-list detour: functions convert the byte array to a flat list of booleans, and inductive proofs operate on this list before translating the results back to the array representation.

This thesis introduces the segment abstraction, where each write operation is recorded as a `(length, value)` pair. Unlike the bit-list approach, the segment abstraction operates at a higher level: it captures write operations as units that mirror the encoder interface. Postconditions are stated in terms of integer-valued segments, so codec-level proofs reason directly about values without decomposing them into individual bits or referring to the underlying buffer.

Composition for `SEQUENCE`

For composite types, the approaches also differ. Bucev et al. use a prefix lemma: if two codec states agree on the buffer contents up to a given bit position, decoding from either state yields the same result. This allows invertibility proofs for a `SEQUENCE` to proceed field by field, with each field relying on the prefix lemma to ignore subsequent fields.

In this thesis, composition is mediated by `segments_eq_lemma`, which establishes that the `segments_of` function is injective: two values that produce the same segment sequence must be equal. Since the postconditions of `encode` and `decode` are stated in terms of `segments_of`, field-by-field composition follows structurally from this injectivity lemma without needing to refer to buffer contents.

Verification Toolchain

Bucev et al. use Stainless [10], targeting a pure functional subset of Scala: imperative code is preprocessed into functional form, eliminating the need for a permission model, and integers are modelled as native bitvectors, so bitwise operations (e.g., shifts) are directly available.

Nagini, in contrast, operates on Python with a mutable heap, where access to each heap location is tracked via fractional permissions, and integers are represented as SMT integers. These differences have direct consequences: the `bytearray`-based `BitStream` required a predicate-based encoding to satisfy Nagini's permission requirements, and the arithmetic encoding of shift operations (described in section 5.2) was necessary because SMT integers do not support shift operations.

Infrastructure Effort

Scala's static type system and Stainless's support for algebraic data types allowed Bucev et al. to focus their engineering effort primarily on the codec proofs. In contrast, Python's dynamic type system and the evolving ecosystem of Python verification tooling necessitated substantial infrastructure work before the codec proofs could begin. As described in Chapter 5, support for `bytearray`, left and right shift operators, dataclasses, and `IntEnum` had to be added to Nagini as part of this project. Additionally, six crashes and three soundness issues were reported to the Nagini maintainers, each accompanied by minimal reproducing test cases, and all were subsequently resolved. Consequently, a larger share of the total effort in this thesis was devoted to enabling infrastructure rather than directly proving codec properties. This investment, however, lays a foundation for future Python verification work using Nagini.

7.4 Practical Benefits of Verification

Beyond the formal correctness guarantees, the verification effort yielded additional practical benefits.

The process of writing formal specifications for the generated code led to the identification of two issues in the constraint-validation logic of the

ASN1SCC Python backend. The `is_constraint_valid` method for `INTEGER` types omitted a lower bound of zero, thereby allowing negative values to be incorrectly accepted as valid. Similarly, the corresponding method for fixed-size `OCTET STRING` types failed to properly enforce the fixed-size constraint on the array length. Both issues were identified and corrected as part of this project.

The verification process also required augmenting the Python code with type annotations compatible with Mypy. These annotations provide static type-checking guarantees independently of the formal proofs, enabling type errors to be detected during development rather than at runtime. As a result, the annotated code is better documented and more amenable to tooling support than unannotated Python.

7.5 Limitations

For completeness, this section summarises the limitations of the current verification scope as described in the preceding chapters.

Among the ASN.1 types supported by the Python backend, `SEQUENCE OF`, `CHOICE`, `BIT STRING`, `IA5String`, and `REAL` are not yet covered. Within the types that are verified, additional restrictions apply. `INTEGER` is limited to constrained ranges whose bounds fit within 64 bits, matching the precondition of the underlying `encode_integer` method. Unconstrained integers are excluded. `OCTET STRING` is restricted to fixed-size encodings without a length prefix; variable-length variants are not verified. `SEQUENCE` does not support optional fields or nested types whose encoding size depends on runtime values. Finally, only the uPER encoding scheme is covered; no ACN-specific encoding methods have been verified.

The verification addresses only the happy path. For encoding, the precondition `is_constraint_valid_pure` restricts the proof to constraint-satisfying values; the behavior of `encode` on invalid inputs is not verified. For decoding, the precondition `segments_valid` assumes that a valid encoding is present at the current read position. The behavior of `decode` on malformed input is similarly outside the scope. A detailed technical discussion of the existing runtime error-handling infrastructure and requirements for extending verification to these cases is provided in section 4.4.

A further practical constraint concerns the generated code itself. ASN1SCC defines nested classes inside each generated class to hold the constants for error codes and messages. Nagini does not currently support nested classes. These definitions must therefore be removed from the generated code before it can be verified.

The coverage limitations described above do not reflect fundamental barriers in the verification architecture. Adding variable-length OCTET STRING and optional SEQUENCE fields would follow the same segment-based pattern already established: encoding would prepend a length or presence segment, and `segments_of` would be extended accordingly. In both cases, the capacity precondition of `encode` generalises to a runtime-dependent value rather than a compile-time constant and can be expressed as a pure function of the value being encoded.

SEQUENCE OF and CHOICE fit naturally into the compositional proof structure used for SEQUENCE: `segments_of` for a SEQUENCE OF value is defined as the concatenation of its elements' segments, while `segments_of` for a CHOICE value selects the segment sequence of the active variant. ACN encoding would require new library methods with their own postconditions, but the segment abstraction and round-trip proof structure carry over unchanged. The remaining uPER codec methods follow the same established pattern. In each case, the primary effort lies in implementing the type-specific auxiliary functions and their associated lemmata, rather than redesigning the proof approach.

7.5.1 Nagini Prerequisites for REAL

Unlike all other omissions discussed above, which can be addressed within the current Nagini infrastructure by introducing additional type-specific auxiliary functions and lemmata, verifying REAL would first require resolving several obstacles at the level of the verification tool itself.

Nagini provides three modes for reasoning about floating-point values. In the default mode, floats are treated as abstract values, and all operations are modelled as uninterpreted functions. This makes it impossible to establish meaningful properties of floating-point arithmetic. The `real` mode represents floats as exact real numbers, which does not align with the IEEE 754 semantics required for the uPER encoding of REAL. The `ieee32` mode provides a faithful IEEE 754 [2] model; however, as noted in the Nagini documentation [1], it often leads to prohibitively long verification times or even non-termination.

The uPER encoding of REAL represents a floating-point value via its IEEE 754 binary form, decomposed into sign, base-2 exponent, and mantissa. In the ASN1SCC Python backend, this decomposition relies on standard library functions such as `math.frexp`, none of which are currently modelled in Nagini. Furthermore, exponentiation is not directly supported by linear arithmetic solvers. Experience with integer bit-shift operations (section 5.2) suggests that handling exponentiation would require a comparable engineering effort to obtain tractable proof obligations.

[1] https://github.com/marcoeilers/nagini

Taken together, these limitations make `REAL` the one unverified type that cannot be supported solely by extending the current framework with additional auxiliary functions and lemmata; instead, it requires substantial extensions to the underlying verification infrastructure.

Chapter 8

Conclusion

This thesis establishes the formal verification of the ASN.1/uPER Python codec generated by ASN1SCC, covering the full pipeline from raw bit-level operations through to round-trip correctness of generated data classes. The central architectural contribution is the segment abstraction, which decouples the bit-level implementation from the codec-level proofs. Encoders and decoders are specified and verified in terms of segment sequences rather than buffer contents, thereby making individual proof obligations tractable and enabling the round-trip argument to be constructed compositionally at the top level.

Building on this foundation, round-trip correctness has been established for the types `BOOLEAN`, `ENUMERATED`, `NULL`, constrained `INTEGER`, fixed-size `OCTET STRING`, and `SEQUENCE` (excluding optional fields or dynamically sized children). The ASN1SCC Python backend has been extended to automatically generate the necessary Nagini annotations.

The current scope covers uPER encoding under the assumption of valid inputs (i.e., the *happy path*). Error handling and invalid-input behavior are not yet verified. The required extensions to Nagini, including support for `bytearray`, bitwise shift operators, dataclasses, and `IntEnum`, expand the range of Python language features that can be formally verified beyond the scope of this project. Due to Python's dynamic type system and still-maturing ecosystem of verification tools, substantially more infrastructure development was required compared to the prior Scala verification effort [5]. As such, the Nagini extensions constitute contributions in their own right, alongside the codec proofs they enable.

These results demonstrate that the class of correctness guarantee established by Bucev et al. for the Scala backend can also be achieved for Python, and that the verification approach scales compositionally to the generated code layer. For safety-critical applications, such as those of the European Space

Agency, this provides a mathematically rigorous correctness guarantee that cannot be achieved through testing alone.

8.1 Future Work

While this thesis establishes a solid foundation for the verification of the Python backend, several directions for future work remain.

8.1.1 Codec Coverage

The most immediate extensions follow the proof pattern established in this thesis and primarily require the implementation of type-specific segment postconditions.

On the runtime library side, the remaining uPER variants — including `BIT STRING`, the bit-pattern, bit-null-terminated string types, and unconstrained integers — fit naturally within this framework. On the data class side, `SEQUENCE OF` and `CHOICE` align with the compositional proof structure established for `SEQUENCE`. Within `SEQUENCE` itself, support for optional fields and dynamically sized nested types remains to be verified.

ACN encoding, which serves as the primary encoding scheme in the prior Scala verification work [5], would require the introduction of new library methods with ACN-specific postconditions. However, the segment abstraction and the overall round-trip proof structure remain directly applicable.

The verification of `REAL` constitutes the only case that cannot be addressed solely through extensions at the codec layer, as it additionally requires enhancements to Nagini's support for floating-point reasoning.

8.1.2 Nagini

As discussed in section 7.5.1, enabling the verification of `REAL` requires improvements to Nagini's capabilities for floating-point reasoning, particularly in modelling standard library functions such as `math.frexp` and handling exponentiation as required by `REAL` encoding.

Extending the preprocessing mode beyond its current single-file limitation to support imported modules is necessary to eliminate the Nagini runtime dependency across a full project. Additionally, an opt-in verification mode, where only selected functions or classes are translated rather than all dependencies, would further reduce verification overhead and support incremental adoption in large codebases.

As discussed in section 6.2, the soundness of `PInt` annotations currently depends on a manual guarantee that annotated functions do not inspect the runtime type of their parameters. Automating this check in Nagini would

eliminate a potential source of unsoundness. Furthermore, a function-level `PInt` annotation, marking all integer parameters of a function as primitive in a single declaration, would be required to enable compatibility with the preprocessing mode.

Support for nested classes would remove the need to manually eliminate error-code constant definitions from the generated code prior to verification. Finally, several usability improvements would benefit larger-scale verification efforts: support for the `Self` type; a mechanism to aggregate multiple frame conditions of the form `Ensures(x == Old(x))` into a single abstraction; and class invariants that are automatically enforced across all methods.

Appendix A

Verification Times

Component	Version / Details
Hardware	
CPU	AMD Ryzen 7 9800X3D, 8 cores @ 4.7 GHz
RAM	64GB, 2x Kingston KF560C30-32
Mainboard	Gigabyte B650 Aorus Elite AX V2
Software	
OS	WSL 2.6.30 with Ubuntu 24.04.1 on Windows 11
Python	3.9.25
Java	OpenJDK 21.0.10
Nagini Versions	
Baseline	`ea43a7ec466a1f39e57f25c60cb8ab87be11ddba`
Shift Benchmark	`8c79915d34498661f8e7d737db7d8772bc7de180`

Table A.1: Benchmark and verification environment.

All verification times were measured on the machine described in Table A.1. Each function was verified individually using `nagini --select`. The reported time corresponds to a single execution, not an average, and reflects the wall-clock time as reported by Nagini.

A.1 Runtime Library

File	Function	Time (s)
`asn1_types.py`	`Asn1Base.class_predicate`	38.5
`asn1_types.py`	`Asn1Base.is_constraint_valid`	37.3
`asn1_types.py`	`Asn1Boolean.__and__`	41.9

File	Function	Time (s)
asn1_types.py	Asn1Boolean.__bool__	37.3
asn1_types.py	Asn1Boolean.__eq__	39.1
asn1_types.py	Asn1Boolean.__ge__	42.2
asn1_types.py	Asn1Boolean.__gt__	42.6
asn1_types.py	Asn1Boolean.__invert__	43.0
asn1_types.py	Asn1Boolean.__le__	40.1
asn1_types.py	Asn1Boolean.__lt__	39.3
asn1_types.py	Asn1Boolean.__ne__	39.4
asn1_types.py	Asn1Boolean.__or__	42.1
asn1_types.py	Asn1Boolean.__xor__	41.7
asn1_types.py	Asn1Boolean.class_predicate	33.5
asn1_types.py	Asn1Boolean.is_constraint_valid	42.7
asn1_types.py	Asn1ConstraintValidResult.__bool__	37.1
asn1_types.py	Asn1ConstraintValidResult.__post_init__	38.6
asn1_types.py	NullType.__bool__	41.0
asn1_types.py	NullType.__eq__	39.8
asn1_types.py	NullType.__ne__	16.8
asn1_types.py	NullType.class_predicate	38.8
asn1_types.py	NullType.decode	16.3
asn1_types.py	NullType.is_constraint_valid	17.2
asn1_types.py	TOTAL	846.1
verification.py	__lemma_byte_read_bit_equal	33.5
verification.py	__lemma_byte_set_bit_false_equal	41.2
verification.py	__lemma_byte_set_bit_true_equal	70.4
verification.py	__lemma_byte_set_bits_prefix	29.8
verification.py	__lemma_byte_set_bits_suffix	40.1
verification.py	__lemma_byte_set_bits_value	57.9
verification.py	__lemma_byteseq_equal_monotonic	37.2
verification.py	__lemma_byteseq_equal_until_transitiv	34.6
verification.py	__lemma_byteseq_set_bit_prefix	25.4
verification.py	__lemma_byteseq_set_bit_value	27.5
verification.py	__lemma_byteseq_set_bits_prefix	26.9
verification.py	__lemma_byteseq_set_bits_value	27.5
verification.py	_lemma_byte_read_bits_equal_monotonic	37.5
verification.py	_lemma_byte_read_bits_induction_lsb	90.4
verification.py	_lemma_byte_set_bit_equal	26.8
verification.py	_lemma_byte_set_bits	28.3
verification.py	_lemma_byteseq_set_bit	23.6
verification.py	byte_read_bit	32.3
verification.py	byte_read_bits	32.7
verification.py	byte_set_bit	27.1
verification.py	byte_set_bits	26.9

File	Function	Time (s)
verification.py	byteseq_bytes_equal_until	32.4
verification.py	byteseq_equal_until	34.2
verification.py	byteseq_read_bit	28.1
verification.py	byteseq_read_bits	28.2
verification.py	byteseq_set_bit	25.4
verification.py	byteseq_set_bits	24.5
verification.py	floor_align_byte	31.8
verification.py	lemma_byteseq_equal_read_bits	36.8
verification.py	lemma_byteseq_read_bits_aligned	31.6
verification.py	lemma_byteseq_read_bits_equal	30.5
verification.py	lemma_byteseq_read_bits_induction_lsb	30.5
verification.py	lemma_byteseq_read_full_byte	29.5
verification.py	lemma_byteseq_set_bits	23.9
verification.py	lemma_byteseq_set_bits_combine	24.8
verification.py	lemma_byteseq_set_bits_eq	24.8
verification.py	TOTAL	1'214.3
segment.py	lemma_byteseq_equal_segments_contained	37.4
segment.py	lemma_segments_byteseq	43.3
segment.py	lemma_segments_byteseq_full	36.3
segment.py	lemma_segments_byteseq_full_equal	31.1
segment.py	lemma_segments_contained_read	32.1
segment.py	lemma_segments_total_length_split	34.2
segment.py	lemma_segments_total_length_uniform	33.0
segment.py	segment_from_byte	27.3
segment.py	segment_invariant	31.1
segment.py	segment_to_byte	28.4
segment.py	segments_add	32.4
segment.py	segments_contained	35.5
segment.py	segments_drop	34.7
segment.py	segments_from_byteseq	333.4
segment.py	segments_from_byteseq_full	33.5
segment.py	segments_invariant	33.9
segment.py	segments_take	34.2
segment.py	segments_to_byteseq	30.5
segment.py	segments_to_byteseq_full	30.5
segment.py	segments_total_length	32.4
segment.py	TOTAL	964.9
bitstream.py	BitStream.__byte_set_bit	24.4
bitstream.py	BitStream.__init__	32.8
bitstream.py	BitStream.__read_bit	25.5
bitstream.py	BitStream.__read_current_bit	27.7

File	Function	Time (s)
bitstream.py	BitStream.__write_bit	26.6
bitstream.py	BitStream.__write_bits32	30.1
bitstream.py	BitStream._increment_segment_read_index	35.0
bitstream.py	BitStream._lemma_read_current_bit_pure	28.3
bitstream.py	BitStream._remaining_bits	26.4
bitstream.py	BitStream._shift_bit_index	28.7
bitstream.py	BitStream.bitstream_invariant	26.5
bitstream.py	BitStream.buffer	24.6
bitstream.py	BitStream.buffer_size	20.2
bitstream.py	BitStream.buffer_size_bits	20.9
bitstream.py	BitStream.current_bit_position	19.8
bitstream.py	BitStream.current_byte_position	20.8
bitstream.py	BitStream.current_used_bits	19.4
bitstream.py	BitStream.current_used_bytes	22.5
bitstream.py	BitStream.from_bitstream	34.4
bitstream.py	BitStream.get_data	26.6
bitstream.py	BitStream.of_size	13.3
bitstream.py	BitStream.position_invariant	26.1
bitstream.py	BitStream.read_align_to_byte	32.5
bitstream.py	BitStream.read_bit	30.4
bitstream.py	BitStream.read_bits	32.3
bitstream.py	BitStream.remaining_bits	22.1
bitstream.py	BitStream.reset	34.9
bitstream.py	BitStream.segments	22.3
bitstream.py	BitStream.segments_predicate	30.9
bitstream.py	BitStream.segments_read_aligned	32.8
bitstream.py	BitStream.segments_read_index	21.2
bitstream.py	BitStream.set_bit_index	27.6
bitstream.py	BitStream.set_position	27.4
bitstream.py	BitStream.to_bit_index	26.0
bitstream.py	BitStream.to_hex_string	21.8
bitstream.py	BitStream.validate_offset	27.1
bitstream.py	BitStream.write_align_to_byte	34.0
bitstream.py	BitStream.write_bit	28.2
bitstream.py	BitStream.write_bits	29.9
bitstream.py	TOTAL	1'042.0
codec.py	Codec.__init__	43.1
codec.py	Codec.bit_index	17.8
codec.py	Codec.buffer	18.0
codec.py	Codec.buffer_size	22.0
codec.py	Codec.codec_predicate	31.1
codec.py	Codec.from_buffer	43.2

File	Function	Time (s)
codec.py	Codec.from_codec	40.8
codec.py	Codec.get_bitstream_buffer	30.1
codec.py	Codec.index_zero	40.6
codec.py	Codec.of_size	43.2
codec.py	Codec.remaining_bits	18.5
codec.py	Codec.segments	16.7
codec.py	Codec.segments_read_index	15.9
codec.py	DecodeResult.__bool__	27.6
codec.py	EncodeResult.__bool__	25.2
codec.py	TOTAL	433.8
decoder.py	Decoder.__read_align	51.4
decoder.py	Decoder._error_float	28.0
decoder.py	Decoder._error_int	26.9
decoder.py	Decoder._error_str	27.8
decoder.py	Decoder.align_to_byte	42.2
decoder.py	Decoder.align_to_dword	42.8
decoder.py	Decoder.align_to_word	43.0
decoder.py	Decoder.current_segment	39.7
decoder.py	Decoder.decode_constrained_pos_whole_number	44.3
decoder.py	Decoder.decode_constrained_whole_number	43.1
decoder.py	Decoder.decode_integer	51.0
decoder.py	Decoder.decode_null	22.8
decoder.py	Decoder.decode_octet_string_no_length	31.5
decoder.py	Decoder.decode_octet_string_no_length_vec	31.9
decoder.py	Decoder.decode_unsigned_integer	45.1
decoder.py	Decoder.has_segments_of_length	43.1
decoder.py	Decoder.read_aligned	42.6
decoder.py	Decoder.read_bit	45.8
decoder.py	Decoder.read_bits	115.2
decoder.py	Decoder.read_byte	46.4
decoder.py	Decoder.read_byte_array	203.3
decoder.py	Decoder.read_has_byte_segments	43.7
decoder.py	Decoder.read_invariant	39.1
decoder.py	TOTAL	1'150.6
encoder.py	Encoder.__write_align	47.8
encoder.py	Encoder.align_to_byte	43.7
encoder.py	Encoder.align_to_dword	38.4
encoder.py	Encoder.align_to_word	40.8
encoder.py	Encoder.append_bit	47.5
encoder.py	Encoder.append_bits	86.9
encoder.py	Encoder.append_byte	46.4

File	Function	Time (s)
encoder.py	Encoder.append_byte_array	111.1
encoder.py	Encoder.encode_constrained_pos_whole_number	40.7
encoder.py	Encoder.encode_constrained_whole_number	40.9
encoder.py	Encoder.encode_integer	49.4
encoder.py	Encoder.encode_null	29.3
encoder.py	Encoder.encode_octet_string_no_length	39.9
encoder.py	Encoder.encode_octet_string_no_length_vec	40.8
encoder.py	Encoder.encode_unsigned_integer	42.3
encoder.py	Encoder.get_decoder	40.4
encoder.py	Encoder.last_segment	40.1
encoder.py	Encoder.write_invariant	41.0
encoder.py	TOTAL	867.3
codec_uper.py	UPEREncoder.get_decoder	19.7
codec_uper.py	TOTAL	19.7

A.2 Definitions for Generated Dataclasses

```
BasicBool ::= BOOLEAN

MyInt ::= INTEGER (0..255)

MyOctetString ::= OCTET STRING (SIZE(4))

Status ::= ENUMERATED {
  idle(0),
  active(1),
  error(2)
}

ShortSequence ::= SEQUENCE {
  flag   BOOLEAN,
  status Status
}

NestedSequence ::= SEQUENCE {
  inner  ShortSequence,
  value  MyInt,
  data   MyOctetString
}
```

Listing 71: ASN.1 definitions used for generated-class verification timing.

Appendix B

Benchmarks

All benchmarks were executed with 10 repetitions per test case using Nagini's built-in `--benchmark` flag and a timeout of 120 seconds per run. The experimental environment is described in Table A.1. The specific Nagini revisions are available in the Nagini repository (https://github.com/marcoeilers/nagini).

B.1 Test Cases

```
from nagini_contracts.contracts import *

MAX_BITOP_LENGTH = 16
NO_OF_BITS_IN_BYTE = 8

@Pure
@Opaque
def byte_read_bit(byte: PInt, position: PInt) -> bool:
    """Read a single bit from a byte. Position starts from MSB."""
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Decreases(None)
    return bool((byte >> (7 - position)) % 2)

@Pure
@Opaque
def byteseq_read_bit(byteseq: PByteSeq, position: PInt) -> bool:
    """Read a single bit from the byte sequence."""
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    byte_position = position // NO_OF_BITS_IN_BYTE
    bit_position = position % NO_OF_BITS_IN_BYTE
    return byte_read_bit(byteseq[byte_position], bit_position)
```

```
@Pure
@Opaque
def byteseq_read_bits(byteseq: PByteSeq, position: PInt, length: PInt) ->
↪  int:
    """Read a number of bits from the byte sequence"""
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(length)
    Ensures(0 <= Result() and Result() < (1 << length))

    if length == 0:
        return 0

    prefix = byteseq_read_bits(byteseq, position, length - 1) << 1
    single = byteseq_read_bit(byteseq, position + length - 1)
    return prefix + single
```

Listing 72: Read-Bit Test Case

```
from nagini_contracts.contracts import *

MAX_BITOP_LENGTH = 16
NO_OF_BITS_IN_BYTE = 8

@Pure
@Opaque
def byte_set_bit(byte: PInt, bit: bool, position: PInt) -> int:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() <= 0xFF)

    if bit:
        return byte | (1 << (7 - position))
    else:
        return byte & ~(1 << (7 - position))

@Pure
@Opaque
def byteseq_set_bit(byteseq: PByteSeq, bit: PBool, position: PInt) ->
↪  PByteSeq:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(len(byteseq) == len(Result()))
    byte_position = position // NO_OF_BITS_IN_BYTE
    bit_position = position % NO_OF_BITS_IN_BYTE
    return byteseq.update(byte_position,
    ↪  byte_set_bit(byteseq[byte_position], bit, bit_position))

@Pure
```

```
@Opaque
def byteseq_set_bits(byteseq: PByteSeq, value: PInt, position: PInt,
↪  length: PInt) -> PByteSeq:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Decreases(length)
    Ensures(Implies(length == 0, Result() == byteseq))
    Ensures(len(byteseq) == len(Result()))

    if length == 0:
        return byteseq

    rec = byteseq_set_bits(byteseq, value >> 1, position, length - 1)
    bit = bool(value % 2)
    new_seq = byteseq_set_bit(rec, bit, position + length - 1)

    return new_seq
```

Listing 73: Set-Bit Test Case

```
from nagini_contracts.contracts import *

NO_OF_BITS_IN_BYTE = 8

@Pure
@Opaque
def byte_read_bits(byte: PInt, position: PInt, length: PInt) -> int:
    """Read up to 8 bits from a byte. Position starts from MSB."""
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= length and length <= NO_OF_BITS_IN_BYTE)
    Requires(0 <= position and position + length <= NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() < (1 << length))
    Ensures(Implies(byte < (1 << length) and position ==
    ↪  NO_OF_BITS_IN_BYTE - length, Result() == byte))
    return (byte >> (NO_OF_BITS_IN_BYTE - position - length)) % (1 <<
    ↪  length)

@Pure
@Opaque
def byte_set_bit(byte: PInt, bit: bool, position: PInt) -> int:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() <= 0xFF)

    if bit:
        return byte | (1 << (7 - position))
    else:
        return byte & ~(1 << (7 - position))
```

```
@Pure
@Opaque
def byte_set_bits(byte: PInt, value: PInt, position: PInt, length: PInt)
↪  -> int:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= length and length <= NO_OF_BITS_IN_BYTE)
    Requires(0 <= position and position + length <= NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Ensures(0 <= Result() and Result() <= 0xFF)

    upper = byte_read_bits(byte, 0, position) << (NO_OF_BITS_IN_BYTE -
    ↪  position)
    middle = value << (NO_OF_BITS_IN_BYTE - position - length)
    lower = byte % (1 << (NO_OF_BITS_IN_BYTE - position - length))
    return upper + middle + lower

@Pure
@Opaque
def _lemma_byte_set_bit_equal(byte: int, bit: bool, position: int) ->
↪  bool:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Ensures(byte_set_bit(byte, bit, position) == byte_set_bits(byte, bit,
    ↪  position, 1))
    Ensures(Result())

    new_byte_a = Reveal(byte_set_bit(byte, bit, position))
    new_byte_b = Reveal(byte_set_bits(byte, bit, position, 1))
    upper_b = Reveal(byte_read_bits(byte, 0, position))

    return new_byte_a == new_byte_b
```

Listing 74: Set-Equal Test Case — Single Lemma

```
from nagini_contracts.contracts import *

NO_OF_BITS_IN_BYTE = 8

@Pure
@Opaque
def byte_read_bits(byte: PInt, position: PInt, length: PInt) -> int:
    """Read up to 8 bits from a byte. Position starts from MSB."""
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= length and length <= NO_OF_BITS_IN_BYTE)
    Requires(0 <= position and position + length <= NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() < (1 << length))
    Ensures(Implies(byte < (1 << length) and position ==
    ↪  NO_OF_BITS_IN_BYTE - length, Result() == byte))
    return (byte >> (NO_OF_BITS_IN_BYTE - position - length)) % (1 <<
    ↪  length)
```

```
@Pure
@Opaque
def byte_set_bit(byte: PInt, bit: bool, position: PInt) -> int:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() <= 0xFF)

    if bit:
        return byte | (1 << (7 - position))
    else:
        return byte & ~(1 << (7 - position))

@Pure
@Opaque
def byte_set_bits(byte: PInt, value: PInt, position: PInt, length: PInt)
↪ -> int:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= length and length <= NO_OF_BITS_IN_BYTE)
    Requires(0 <= position and position + length <= NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Ensures(0 <= Result() and Result() <= 0xFF)

    upper = byte_read_bits(byte, 0, position) << (NO_OF_BITS_IN_BYTE -
    ↪  position)
    middle = value << (NO_OF_BITS_IN_BYTE - position - length)
    lower = byte % (1 << (NO_OF_BITS_IN_BYTE - position - length))
    return upper + middle + lower

@Pure
@Opaque
def __lemma_byte_set_bit_true_equal(byte: int, position: int) -> bool:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Ensures(byte_set_bit(byte, True, position) == byte_set_bits(byte, True,
    ↪  position, 1))
    Ensures(Result())

    new_byte_a = Reveal(byte_set_bit(byte, True, position))
    new_byte_b = Reveal(byte_set_bits(byte, True, position, 1))
    upper_b = Reveal(byte_read_bits(byte, 0, position))

    return new_byte_a == new_byte_b

@Pure
@Opaque
def __lemma_byte_set_bit_false_equal(byte: int, position: int) -> bool:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Ensures(byte_set_bit(byte, False, position) == byte_set_bits(byte,
    ↪  False, position, 1))
    Ensures(Result())

    new_byte_a = Reveal(byte_set_bit(byte, False, position))
```

```
    new_byte_b = Reveal(byte_set_bits(byte, False, position, 1))
    upper_b = Reveal(byte_read_bits(byte, 0, position))

    return new_byte_a == new_byte_b

# Splitting up the verification seems to improve performance
@Pure
@Opaque
def _lemma_byte_set_bit_equal(byte: int, bit: bool, position: int) ->
↪  bool:
    Requires(0 <= byte and byte <= 0xFF)
    Requires(0 <= position and position < NO_OF_BITS_IN_BYTE)
    Ensures(byte_set_bit(byte, bit, position) == byte_set_bits(byte, bit,
    ↪  position, 1))
    Ensures(Result())

    if bit:
        return __lemma_byte_set_bit_true_equal(byte, position)
    else:
        return __lemma_byte_set_bit_false_equal(byte, position)
```

Listing 75: Set-Equal Test Case — Split Lemmata

```
from nagini_contracts.contracts import *

NO_OF_BITS_IN_BYTE = 8
MAX_BITOP_LENGTH = 32

#region Byteseq equality

@Pure
@Opaque
@ContractOnly
def byteseq_equal_until(b1: PByteSeq, b2: PByteSeq, end: PInt) -> bool:
    Requires(0 <= end and end <= len(b1) * NO_OF_BITS_IN_BYTE and end <=
    ↪  len(b2) * NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(Implies(b1 == b2, Result()))

@Pure
@Opaque
@ContractOnly
def __lemma_byteseq_equal_monotonic(b1: PByteSeq, b2: PByteSeq, end: int,
↪  prefix: int) -> bool:
    Requires(0 <= prefix and prefix <= end and end <= len(b1) *
    ↪  NO_OF_BITS_IN_BYTE and end <= len(b2) * NO_OF_BITS_IN_BYTE)
    Requires(byteseq_equal_until(b1, b2, end))
    Decreases(None)
    Ensures(byteseq_equal_until(b1, b2, prefix))
    Ensures(Result())

@Pure
```

```
@Opaque
@ContractOnly
def __lemma_byteseq_equal_until_transitiv(b1: PByteSeq, b2: PByteSeq, b3:
↪  PByteSeq, end: int) -> bool:
    Requires(len(b1) <= len(b2))
    Requires(len(b2) <= len(b3))
    Requires(0 <= end and end <= len(b1) * NO_OF_BITS_IN_BYTE)
    Requires(byteseq_equal_until(b1, b2, end))
    Requires(byteseq_equal_until(b2, b3, end))
    Decreases(None)
    Ensures(byteseq_equal_until(b1, b3, end))
    Ensures(Result())

#endregion
#region Read byteseq

@Pure
@Opaque
@ContractOnly
def byteseq_read_bit(byteseq: PByteSeq, position: PInt) -> bool:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)

@Pure
@Opaque
@ContractOnly
def byteseq_read_bits(byteseq: PByteSeq, position: PInt, length: PInt) ->
↪  int:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() < (1 << length))

@Pure
@Opaque
@ContractOnly
def lemma_byteseq_read_bits_equal(byteseq: PByteSeq, position: int) ->
↪  bool:
    Requires(0 <= position and position + 1 <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(byteseq_read_bits(byteseq, position, 1) ==
    ↪  int(byteseq_read_bit(byteseq, position)))
    Ensures(Result())

@Pure
@Opaque
@ContractOnly
def lemma_byteseq_read_bits_induction_lsb(byteseq: PByteSeq, position: int,
↪  length: int) -> bool:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
```

```
    Decreases(None)
    Ensures(Implies(length >= 1, byteseq_read_bits(byteseq, position,
    ↪  length) ==
                                (byteseq_read_bits(byteseq, position,
                                ↪  length - 1) << 1) +
                                (byteseq_read_bits(byteseq, position +
                                ↪  length - 1, 1))))
    Ensures(Result())

@Pure
@Opaque
@ContractOnly
def lemma_byteseq_equal_read_bits(b1: PByteSeq, b2: PByteSeq, equal_end:
↪  int, position: int, length: int) -> bool:
    Requires(0 <= equal_end and equal_end <= len(b1) * NO_OF_BITS_IN_BYTE
    ↪  and equal_end <= len(b2) * NO_OF_BITS_IN_BYTE)
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= equal_end)
    Requires(byteseq_equal_until(b1, b2, equal_end))
    Decreases(None)
    Ensures(byteseq_read_bits(b1, position, length) ==
    ↪  byteseq_read_bits(b2, position, length))

#endregion
#region Set byteseq

@Pure
@Opaque
@ContractOnly
def byteseq_set_bit(byteseq: PByteSeq, bit: PBool, position: PInt) ->
↪  PByteSeq:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(len(byteseq) == len(Result()))

@Pure
@Opaque
def byteseq_set_bits(byteseq: PByteSeq, value: PInt, position: PInt,
↪  length: PInt) -> PByteSeq:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Decreases(length)
    Ensures(Implies(length == 0, Result() == byteseq))
    Ensures(len(byteseq) == len(Result()))

    if length == 0:
        return byteseq
    rec = byteseq_set_bits(byteseq, value >> 1, position, length - 1)
    bit = bool(value % 2)
    return byteseq_set_bit(rec, bit, position + length - 1)

@Pure
```

```
@Opaque
@ContractOnly
def _lemma_byteseq_set_bit(byteseq: PByteSeq, bit: bool, position: int) ->
↪  bool:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(byteseq_read_bit(byteseq_set_bit(byteseq, bit, position),
    ↪  position) == bit)
    Ensures(byteseq_equal_until(byteseq_set_bit(byteseq, bit, position),
    ↪  byteseq, position))
    Ensures(Result())

# Single merged lemma: prefix and value proved together in one recursive
↪  function
@Pure
@Opaque
def lemma_byteseq_set_bits(byteseq: PByteSeq, value: int, position: int,
↪  length: int) -> bool:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Decreases(length)
    Ensures(byteseq_equal_until(byteseq_set_bits(byteseq, value, position,
    ↪  length), byteseq, position))
    Ensures(byteseq_read_bits(byteseq_set_bits(byteseq, value, position,
    ↪  length), position, length) == value)
    Ensures(Result())

    if length == 0:
        return byteseq_set_bits(byteseq, value, position, length) ==
        ↪  byteseq

    full_seq = Reveal(byteseq_set_bits(byteseq, value, position, length))
    rec_seq = byteseq_set_bits(byteseq, value >> 1, position, length - 1)
    lemma_rec = lemma_byteseq_set_bits(byteseq, value >> 1, position,
    ↪  length - 1)

    Assert(byteseq_equal_until(rec_seq, byteseq, position))
    Assert(byteseq_read_bits(rec_seq, position, length - 1) == value >> 1)

    bit = bool(value % 2)
    set_position = position + length - 1
    new_seq = byteseq_set_bit(rec_seq, bit, set_position)

    lemma_set_bit = _lemma_byteseq_set_bit(rec_seq, bit, set_position)

    # Value proof
    lemma_read_equal = lemma_byteseq_read_bits_equal(new_seq,
    ↪  set_position)
    Assert(byteseq_read_bits(new_seq, set_position, 1) == int(bit))
    lemma_read_equality = lemma_byteseq_equal_read_bits(new_seq, rec_seq,
    ↪  set_position, position, length - 1)
    Assert(byteseq_read_bits(new_seq, position, length - 1) == value >> 1)
```

```
    lemma_read_induction = lemma_byteseq_read_bits_induction_lsb(new_seq,
    ↪  position, length)

    # Prefix proof
    lemma_equal_monotonic = __lemma_byteseq_equal_monotonic(new_seq,
    ↪  rec_seq, position + length - 1, position)
    Assert(byteseq_equal_until(new_seq, rec_seq, position))
    lemma_equal_transitiv = __lemma_byteseq_equal_until_transitiv(new_seq,
    ↪  rec_seq, byteseq, position)

    return byteseq_equal_until(new_seq, byteseq, position) and
    ↪  byteseq_read_bits(new_seq, position, length) == value

#endregion
```

Listing 76: Byteseq-Set-Bits Test Case — Single Lemma

```
from nagini_contracts.contracts import *

NO_OF_BITS_IN_BYTE = 8
MAX_BITOP_LENGTH = 32

#region Byteseq equality

@Pure
@Opaque
@ContractOnly
def byteseq_equal_until(b1: PByteSeq, b2: PByteSeq, end: PInt) -> bool:
    Requires(0 <= end and end <= len(b1) * NO_OF_BITS_IN_BYTE and end <=
    ↪  len(b2) * NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(Implies(b1 == b2, Result()))

@Pure
@Opaque
@ContractOnly
def __lemma_byteseq_equal_monotonic(b1: PByteSeq, b2: PByteSeq, end: int,
↪  prefix: int) -> bool:
    Requires(0 <= prefix and prefix <= end and end <= len(b1) *
    ↪  NO_OF_BITS_IN_BYTE and end <= len(b2) * NO_OF_BITS_IN_BYTE)
    Requires(byteseq_equal_until(b1, b2, end))
    Decreases(None)
    Ensures(byteseq_equal_until(b1, b2, prefix))
    Ensures(Result())

@Pure
@Opaque
@ContractOnly
def __lemma_byteseq_equal_until_transitiv(b1: PByteSeq, b2: PByteSeq, b3:
↪  PByteSeq, end: int) -> bool:
    Requires(len(b1) <= len(b2))
    Requires(len(b2) <= len(b3))
    Requires(0 <= end and end <= len(b1) * NO_OF_BITS_IN_BYTE)
    Requires(byteseq_equal_until(b1, b2, end))
    Requires(byteseq_equal_until(b2, b3, end))
    Decreases(None)
    Ensures(byteseq_equal_until(b1, b3, end))
    Ensures(Result())

#endregion
#region Read byteseq

@Pure
@Opaque
@ContractOnly
def byteseq_read_bit(byteseq: PByteSeq, position: PInt) -> bool:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
```

```
@Pure
@Opaque
@ContractOnly
def byteseq_read_bits(byteseq: PByteSeq, position: PInt, length: PInt) ->
↪  int:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(0 <= Result() and Result() < (1 << length))

@Pure
@Opaque
@ContractOnly
def lemma_byteseq_read_bits_equal(byteseq: PByteSeq, position: int) ->
↪  bool:
    Requires(0 <= position and position + 1 <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(byteseq_read_bits(byteseq, position, 1) ==
    ↪  int(byteseq_read_bit(byteseq, position)))
    Ensures(Result())

@Pure
@Opaque
@ContractOnly
def lemma_byteseq_read_bits_induction_lsb(byteseq: PByteSeq, position: int,
↪  length: int) -> bool:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(Implies(length >= 1, byteseq_read_bits(byteseq, position,
    ↪  length) ==
                                (byteseq_read_bits(byteseq, position,
                                ↪  length - 1) << 1) +
                                (byteseq_read_bits(byteseq, position +
                                ↪  length - 1, 1))))
    Ensures(Result())

@Pure
@Opaque
@ContractOnly
def lemma_byteseq_equal_read_bits(b1: PByteSeq, b2: PByteSeq, equal_end:
↪  int, position: int, length: int) -> bool:
    Requires(0 <= equal_end and equal_end <= len(b1) * NO_OF_BITS_IN_BYTE
    ↪  and equal_end <= len(b2) * NO_OF_BITS_IN_BYTE)
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= equal_end)
    Requires(byteseq_equal_until(b1, b2, equal_end))
    Decreases(None)
    Ensures(byteseq_read_bits(b1, position, length) ==
    ↪  byteseq_read_bits(b2, position, length))

#endregion
```

```
#region Set byteseq

@Pure
@Opaque
@ContractOnly
def byteseq_set_bit(byteseq: PByteSeq, bit: PBool, position: PInt) ->
↪  PByteSeq:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(len(byteseq) == len(Result()))

@Pure
@Opaque
def byteseq_set_bits(byteseq: PByteSeq, value: PInt, position: PInt,
↪  length: PInt) -> PByteSeq:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Decreases(length)
    Ensures(Implies(length == 0, Result() == byteseq))
    Ensures(len(byteseq) == len(Result()))

    if length == 0:
        return byteseq
    rec = byteseq_set_bits(byteseq, value >> 1, position, length - 1)
    bit = bool(value % 2)
    return byteseq_set_bit(rec, bit, position + length - 1)

@Pure
@Opaque
@ContractOnly
def _lemma_byteseq_set_bit(byteseq: PByteSeq, bit: bool, position: int) ->
↪  bool:
    Requires(0 <= position and position < len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Decreases(None)
    Ensures(byteseq_read_bit(byteseq_set_bit(byteseq, bit, position),
    ↪  position) == bit)
    Ensures(byteseq_equal_until(byteseq_set_bit(byteseq, bit, position),
    ↪  byteseq, position))
    Ensures(Result())

# Split lemmata: prefix and value proved separately, combined at top level
@Pure
@Opaque
def __lemma_byteseq_set_bits_prefix(byteseq: PByteSeq, value: int,
↪  position: int, length: int) -> bool:
    """Proof that byteseq_set_bits() preserves previous bits in the
    ↪  sequence."""
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
```

```
    Decreases(length)
    Ensures(byteseq_equal_until(byteseq_set_bits(byteseq, value, position,
    ↪  length), byteseq, position))
    Ensures(Result())

    if length == 0:
        return byteseq_set_bits(byteseq, value, position, length) ==
        ↪  byteseq
    full_seq = Reveal(byteseq_set_bits(byteseq, value, position, length))
    rec_seq = byteseq_set_bits(byteseq, value >> 1, position, length - 1)
    lemma_rec = __lemma_byteseq_set_bits_prefix(byteseq, value >> 1,
    ↪  position, length - 1)
    Assert(byteseq_equal_until(rec_seq, byteseq, position))
    bit = bool(value % 2)
    new_seq = byteseq_set_bit(rec_seq, bit, position + length - 1)
    lemma_set_bit = _lemma_byteseq_set_bit(rec_seq, bit, position + length
    ↪  - 1)
    lemma_equal_monotonic = __lemma_byteseq_equal_monotonic(new_seq,
    ↪  rec_seq, position + length - 1, position)
    Assert(byteseq_equal_until(new_seq, rec_seq, position))
    lemma_equal_transitiv = __lemma_byteseq_equal_until_transitiv(new_seq,
    ↪  rec_seq, byteseq, position)
    return byteseq_equal_until(new_seq, byteseq, position)

@Pure
@Opaque
def __lemma_byteseq_set_bits_value(byteseq: PByteSeq, value: int, position:
↪  int, length: int) -> bool:
    """Proof that byteseq_set_bits() writes the input value."""
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Decreases(length)
    Ensures(byteseq_read_bits(byteseq_set_bits(byteseq, value, position,
    ↪  length), position, length) == value)
    Ensures(Result())

    if length == 0:
        return True
    full_seq = Reveal(byteseq_set_bits(byteseq, value, position, length))
    rec_seq = byteseq_set_bits(byteseq, value >> 1, position, length - 1)
    lemma_rec = __lemma_byteseq_set_bits_value(byteseq, value >> 1,
    ↪  position, length - 1)
    Assert(byteseq_read_bits(rec_seq, position, length - 1) == value >> 1)
    bit = bool(value % 2)
    set_position = position + length - 1
    new_seq = byteseq_set_bit(rec_seq, bit, set_position)
    lemma_set_bit = _lemma_byteseq_set_bit(rec_seq, bit, set_position)
    lemma_read_equal = lemma_byteseq_read_bits_equal(new_seq,
    ↪  set_position)
    Assert(byteseq_read_bits(new_seq, set_position, 1) == int(bit))
    lemma_read_equality = lemma_byteseq_equal_read_bits(new_seq, rec_seq,
    ↪  set_position, position, length - 1)
    Assert(byteseq_read_bits(new_seq, position, length - 1) == value >> 1)
```

```
    lemma_read_induction = lemma_byteseq_read_bits_induction_lsb(new_seq,
    ↪  position, length)
    return byteseq_read_bits(new_seq, position, length) == value

@Pure
@Opaque
def lemma_byteseq_set_bits(byteseq: PByteSeq, value: int, position: int,
↪  length: int) -> bool:
    Requires(0 <= length and length <= MAX_BITOP_LENGTH)
    Requires(0 <= position and position + length <= len(byteseq) *
    ↪  NO_OF_BITS_IN_BYTE)
    Requires(0 <= value and value < (1 << length))
    Decreases(None)
    Ensures(byteseq_equal_until(byteseq_set_bits(byteseq, value, position,
    ↪  length), byteseq, position))
    Ensures(byteseq_read_bits(byteseq_set_bits(byteseq, value, position,
    ↪  length), position, length) == value)
    Ensures(Result())

    lemma_prefix = __lemma_byteseq_set_bits_prefix(byteseq, value,
    ↪  position, length)
    lemma_value = __lemma_byteseq_set_bits_value(byteseq, value, position,
    ↪  length)
    return lemma_prefix and lemma_value

#endregion
```

Listing 77: Byteseq-Set-Bits Test Case — Split Lemmata

Bibliography

[1] ECSS-E-ST-70-41C: Space Engineering — Telemetry and Telecommand Packet Utilization. Technical report, European Cooperation for Space Standardization (ECSS), April 2016.

[2] IEEE Standard for Floating-Point Arithmetic. Technical Report 754-2019, Institute of Electrical and Electronics Engineers, July 2019.

[3] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. The prusti project: Formal verification for rust. In *NASA Formal Methods (14th International Symposium)*, pages 88–108, Berlin, Heidelberg, 2022. Springer-Verlag.

[4] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). https://www.SMT-LIB.org, 2016.

[5] Mario Bucev, Samuel Chassot, Simon Felix, Filip Schramka, and Viktor Kunčak. Formally verifiable generated ASN.1/ACN encoders and decoders: A case study, 2024.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Marco Eilers. *Modular Specification and Verification of Security Properties for Mainstream Languages*. PhD thesis, ETH Zurich, 2022.

[8] Marco Eilers and Peter Müller. Nagini: A static verifier for python. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 596–603, Cham, 2018. Springer International Publishing.

[9] Jad Hamza, Simon Felix, Viktor Kunčak, Ivo Nussbaumer, and Filip Schramka. From verified Scala to STIX file system embedded code using Stainless. In *NASA Formal Methods (14th International Symposium)*, pages 393–410, Berlin, Heidelberg, 2022. Springer-Verlag.

[10] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: formalized foundations for the Stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[11] Jukka Lehtosalo et al. Mypy - optional static typing for Python. https://mypy-lang.org/, 2026.

[12] George Mamais, Thanassis Tsiodras, David Lesens, and Maxime Perrotin. An ASN.1 compiler for embedded/space systems. In *ERTS 2012 proceedings*, Toulouse, France, February 2012.

[13] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[14] International Telecommunication Union Telecommunication Standardization Sector. *X.680: Information Technology - Abstract Syntax Notation One (ASN.1)*. February 2021.

[15] International Telecommunication Union Telecommunication Standardization Sector. *X.691: Information technology - ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*. February 2021.

[16] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems - TOPLAS*, 34:1–58, 04 2012.

[17] N7 Space. Asn.1 pus-c types library. https://n7space.github.io/asn1-pusc-lib/, 2026.

[18] TASTE Project. Technical topic: ASN.1 — An introduction to ACN, 2024. Last edited August 6, 2024.

[19] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 367–379. Springer International Publishing, 2021.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. In consultation with the supervisor, one of the following three options must be selected:

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

○ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used and cited generative artificial intelligence technologies[2].

◉ I confirm that I authored the work in question independently and in my own words, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used generative artificial intelligence technologies[3]. In consultation with the supervisor, I did not cite them.

Title of paper or thesis:

Formally Verified ASN.1 Python Encoders and Decoders

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Schafroth	Luca

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guide.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Würenlingen, 23.03.2026	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] E.g. ChatGPT, DALL E 2, Google Bard
[2] E.g. ChatGPT, DALL E 2, Google Bard
[3] E.g. ChatGPT, DALL E 2, Google Bard