To Space and Back: Verified Serialisation

Formally Verifiable Generated ASN.1/ACN Encoders and Decoders:
A Case Study

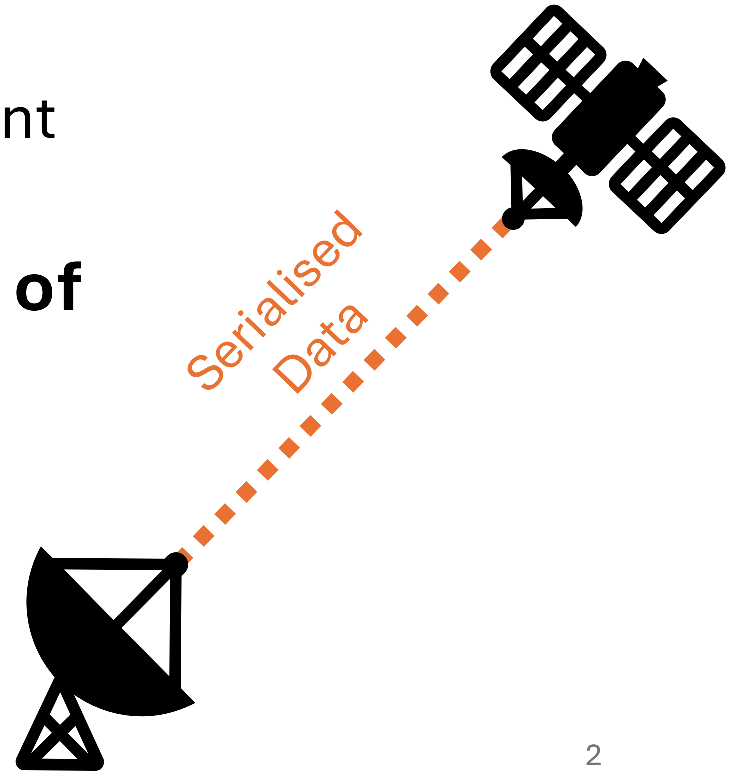
Mario Bucev, **Samuel Chassot**, Simon Felix, Filip Schramka, and Viktor Kunčak

Introduction

Motivation

- Space exploration needs serialisers for communication
- Writing by hand is hard and bug-prone
  - E.g., Solar orbiter: endianness mismatch and different paddings required patching after launch
- Correctness is critical**: bugs could lead to loss of data or vessel**

- → **Generate the code**
- Compile from **ASN.1 format**



Introduction

ASN.1 (Abstract Syntax Notation One)

- Describe datastructures for serialisation
- **Different binary encodings** (e.g., **ACN,** BER, PUR, EPUR, DER)
- **ACN**: customise the binary format → **to support legacy formats**
- Widely used in telecommunications, **notably HTTPS certificates and 5G protocols**

Several ESA missions use **ASN1SCC** compiler

- Takes ASN.1 description and generates code for encoding & decoding

ASN.1 Example: Abstract Syntax

```
MY-MODULE DEFINITIONS AUTOMATIC TAGS ::= BEGIN

IdentifierType ::= INTEGER (-32768..32767)   -- 16 bits

Message ::= SEQUENCE {
    msgId           IdentifierType,
    myflag          INTEGER (1..255),        -- constrained range
    value           REAL,                    -- floating-point
    szDescription   OCTET STRING (SIZE(10))  -- 10 bytes
}

END
```

ACN Example: (Concrete) Binary Format

```
MY-MODULE DEFINITIONS ::= BEGIN

IdentifierType [size 16, encoding twos-complement, endianness little]

Message [] {
  msgId [],
  myflag [size 9, encoding pos-int, align-to-next dword], -- aligned to 32 bits
  value [encoding IEEE754-1985-64],
  szDescription []
}

END
```

ASN1SCC Background

Compiler for ASN.1/ACN format

- Backends for C and Ada

Generated code: combination of static **primitives**

- Datastructure: **BitStream**
- Encoder/decoder for basic types (Codecs)

→ We added a **verified Scala backend**

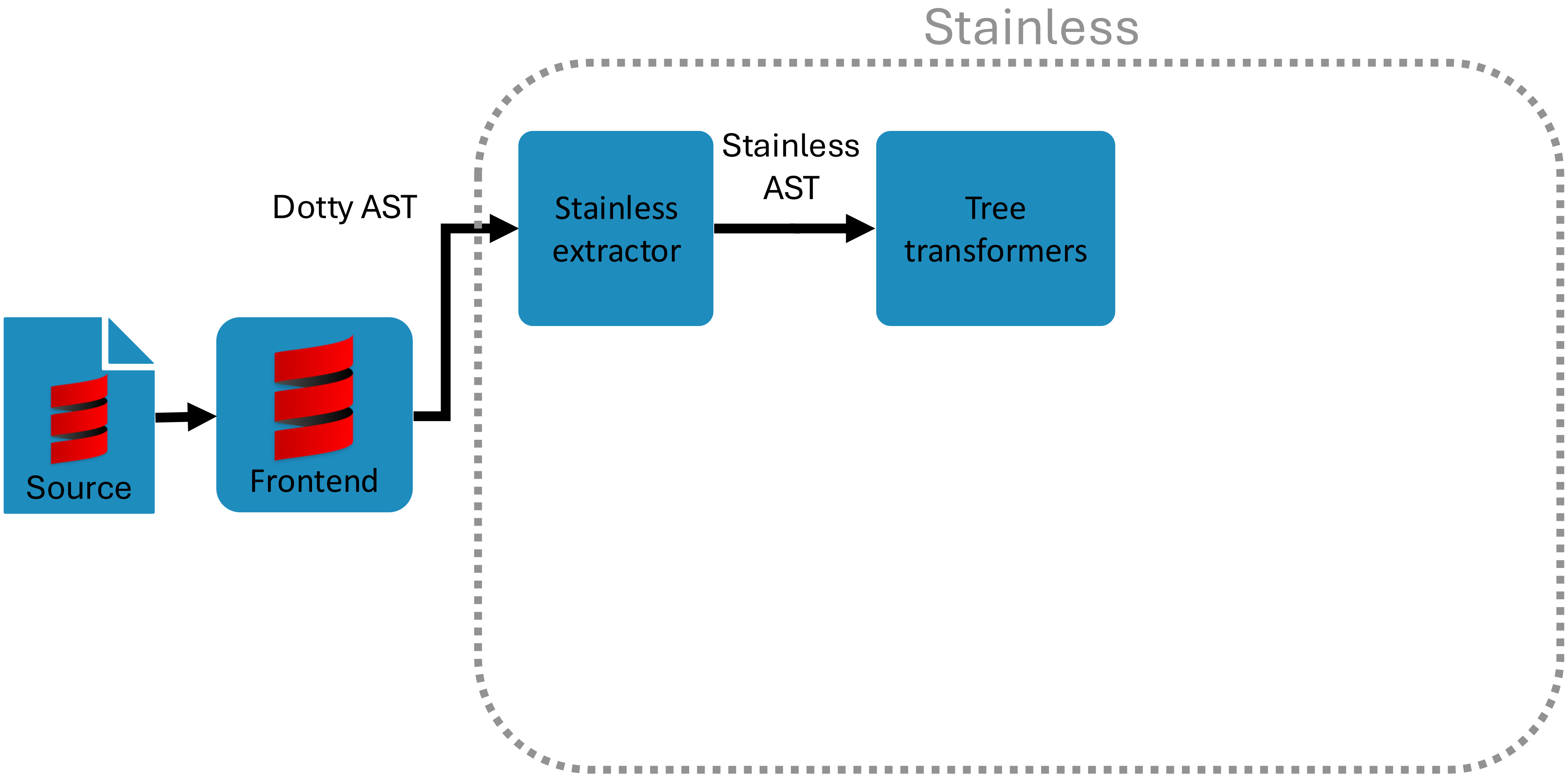
Stainless: Automated Proof

```scala
def zip(xs: List[Int], ys: List[Boolean]): List[(Int, Boolean)] = {
  require(xs.size <= ys.size)
  (xs, ys) match
    case (Cons(x, xs0), Cons(y, ys0)) =>
           Cons((x, y), zip(xs0, ys0))

    case _ => Nil()
}.ensuring (res => res.map(p => p._1) == xs)
```

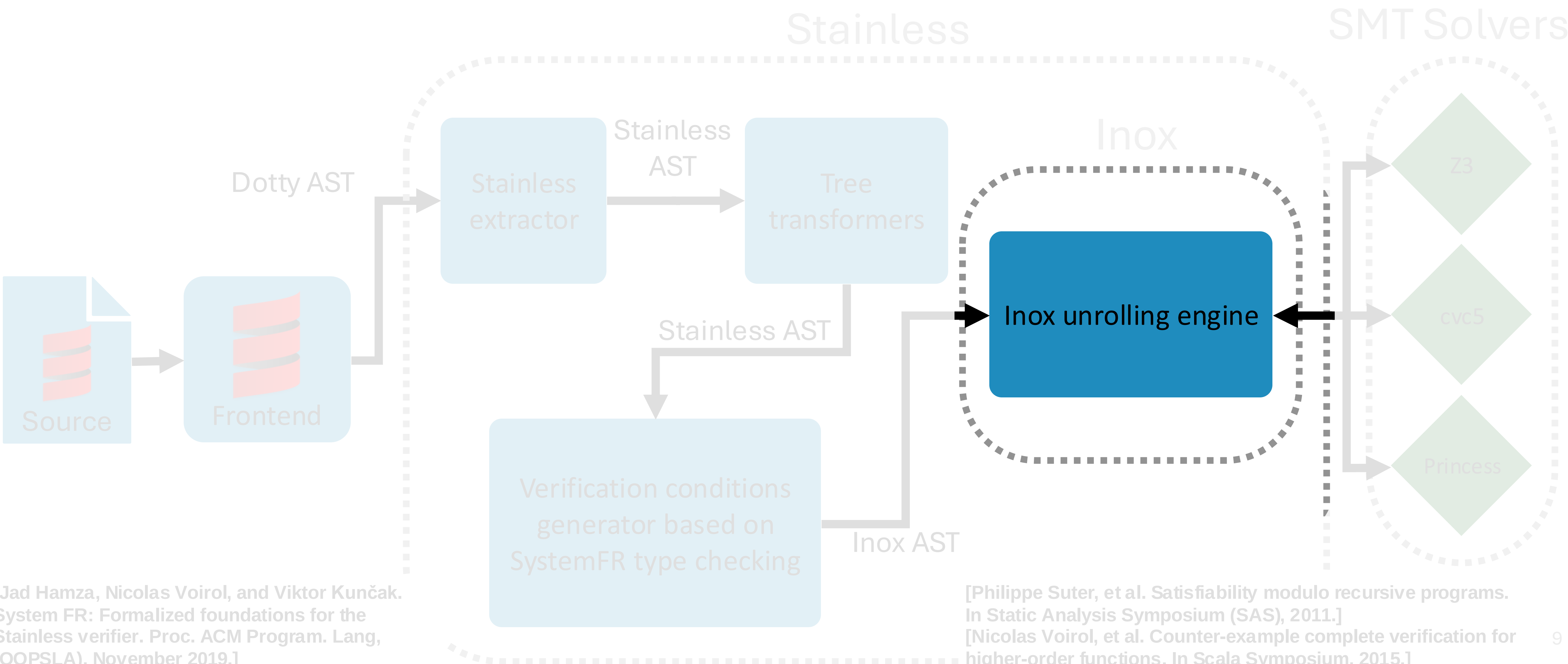
```
stainless summary

zip.scala:5:5:    zip  non-negative measure                          valid  U:smt-cvc5  0.0
zip.scala:9:11:   zip  postcondition                                 valid  U:smt-cvc5  0.1
zip.scala:9:24:   zip  measure decreases                             valid  U:smt-cvc5  0.0
zip.scala:9:24:   zip  precond. (call zip((scrut._1)... @DropVCs , (scrut._...)  valid  U:smt-cvc5  0.1
zip.scala:11:15:  zip  postcondition                                 valid  U:smt-cvc5  0.0

total: 5    valid: 5    (0 from cache, 0 trivial) invalid: 0    unknown: 0    time:    0.29
```

```
warning: Found counter-example:
warning:   xs: List[Int]       -> Cons[Int](0, Nil[Int]())
           ys: List[Boolean] -> Nil[Boolean]()
```

Stainless: verification framework for Scala



Stainless: verification framework for Scala



[Jad Hamza, Nicolas Voirol, and Viktor Kunčak. System FR: Formalized foundations for the Stainless verifier. Proc. ACM Program. Lang, (OOPSLA), November 2019.]

[Philippe Suter, et al. Satisfiability modulo recursive programs. In Static Analysis Symposium (SAS), 2011.]
[Nicolas Voirol, et al. Counter-example complete verification for higher-order functions. In Scala Symposium, 2015.]

Verification approach

2 main steps

1. **Runtime safety** (no crashes, termination) (levels 1, 2, and 3 in the paper)
    - **Code accepted by Stainless**
    - **Automatically generated verification conditions** (e.g., termination, in-bound accesses, overflows, division by zero, casts)
    - To prove in-bound accesses → **add specification about how many bits written/read**
2. **Semantic correctness** (level 4 in the paper)
    - Add specification about **invertibility**

For static primitives and generated code

Verification: Static Primitives

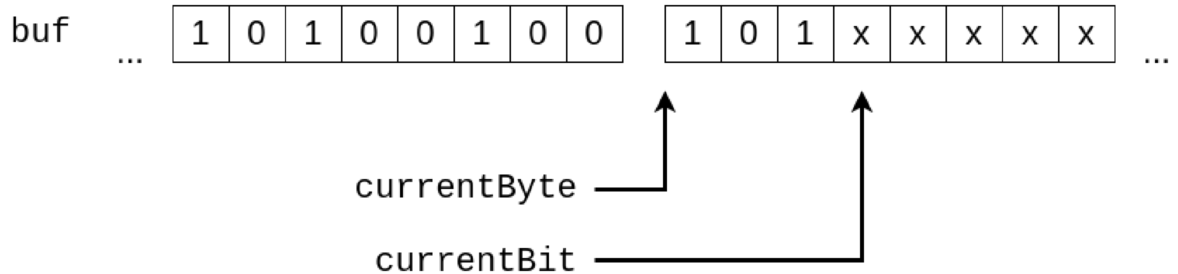
BitStream Datastructure

Codec ACN

BitStream: Background

- Datastructure to represent a stream of bits for decoding and encoding
- Mutable array of bytes, with a moving cursor
- Offers operations to read and write at the bit and at the byte level

Bit-Level Operations	
Encoding Functions	**Decoding Functions**
appendBit, appendBitOne, appendBitZero	readBit
appendNBits, appendNZeroBits, appendNOneBits	readBits
appendBitFromByte	readBit
appendBitsLSBFirst	readNBitsLSBFirst
appendLSBBitsMSBFirst	readNLSBBitsMSBFirst
appendBitsMSBFirst	readBits, peekBit
Byte-Level Operations	
appendPartialByte	readPartialByte
appendByte	readByte
appendByteArray	readByteArray



BitStream: Verification

Step 0: Write the Scala backend BitStream

- Translation of the C backend
- With test suites

Step 1: Prove runtime safety

- Refactor to conform to the Scala fragment supported by Stainless
- Prove **in-bound accesses**, overflow absence, termination

→ Add specification & proof: **number of bits written/read by each function**

BitStream: appendBitsMSBFirst Example

```
def appendBitsMSBFirst(srcBuffer: Array[UByte], nBits: Long, from: Long = 0): Unit = {
  require(nBits >= 0)
  require(from >= 0)
  require(from < Long.MaxValue - nBits)
  require(nBits + from <= srcBuffer.length.toLong * 8L)
  require(BitStream.validate_offset_bits(buf.length.toLong, currentByte.toLong, currentBit.toLong, nBits))
  //
}.ensuring(_ => // omitted: buffer length preserved
      BitStream.bitIndex(buf.length, currentByte, currentBit) ==
      BitStream.bitIndex(old(this).buf.length, old(this).currentByte, old(this).currentBit) + nBits
  )
```

BitStream: Verification

Step 2: semantic correctness

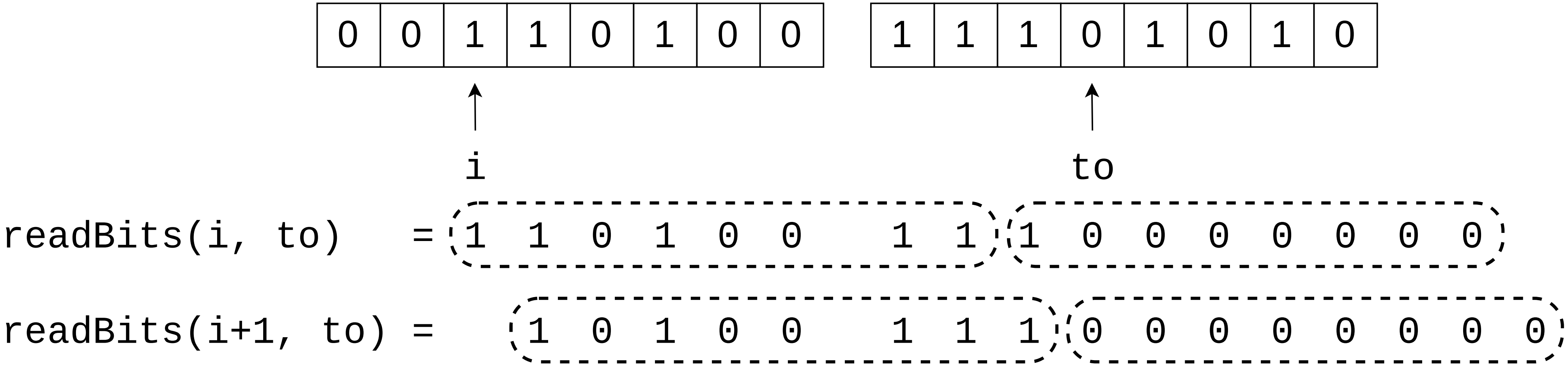
- Prove the invertibility
- i.e., decoding after encoding in the bit stream reads the written value

BitStream: appendBitsMSBFirst Example

```scala
def appendBitsMSBFirst(srcBuffer: Array[UByte], nBits: Long, from: Long = 0): Unit = {
      appendBitsMSBFirstLoop(srcBuffer, from, from + nBits) // Loop as tail rec func
}.ensuring(_ => // ...
  val (r1, r2) = reader(old(this), this)
  val vGot = r1.readBits(nBits)
  byteArrayBitContentSame(srcBuffer, vGot, from, 0, nBits)
)
```

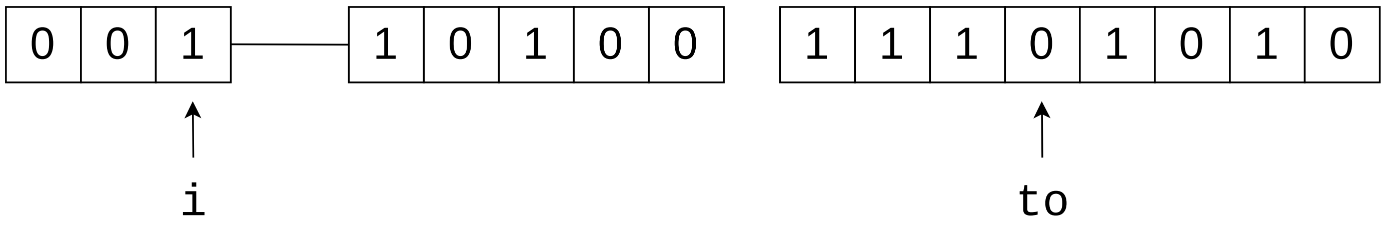
appendBitsMSBFirst: Induction Hypothesis

- Invertibility of appendBitsMSBFirst proved **by induction**
- **Array of bytes → difficult to apply induction hypothesis**



- Not an issue to prove the runtime safety as we proved only how many bits were written

Detour: List of Bits




```
rdBitsLst(i, to)   = Cons(1,List(1, 0, 1, 0, 0,   1, 1, 1, 0, 0, 0, 0, 0, 0, 0))

rdBitsLst(i+1, to) =        List(1, 0, 1, 0, 0,   1, 1, 1, 0, 0, 0, 0, 0, 0, 0)
```

- Applying the IH is natural
- Then prove: same list of Booleans => same bits read in array
- → **Inductive structures better suited for inductive proofs**

ACN Codec Primitives

Also primitive encoding/decoding functions

- E.g.: 64-bits integer in interval [min, max]

```
def encodeConstrainedPosWholeNumber(v: ULong, min: ULong, max: ULong): Unit
def decodeConstrainedPosWholeNumber(min: ULong, max: ULong): ULong
```

Verification: same approach

Relies heavily on BitStream correctness

- IEEE754 real numbers not verified (no float support in Stainless)
- Strings related functions not verified for invertibility, rarely used

Loop Unrolling Trick

```
def uint2int(v: ULong, uintSizeInBytes: Int): Long = {
    require(uintSizeInBytes >= 1 && uintSizeInBytes <= 9)
  // ...
}
```

- Converts unsigned integer to signed one, considering only uintSizeInBytes bytes

~~Hard invariant~~

```
var i: Int = 7
while i >= uintSizeInBytes do
  vv |= ber_aux(i)
  i -= 1
```

→

Automatic proof

```
if(uintSizeInBytes <= 7) then vv |= ber_aux(7)
if(uintSizeInBytes <= 6) then vv |= ber_aux(6)
if(uintSizeInBytes <= 5) then vv |= ber_aux(5)
if(uintSizeInBytes <= 4) then vv |= ber_aux(4)
if(uintSizeInBytes <= 3) then vv |= ber_aux(3)
if(uintSizeInBytes <= 2) then vv |= ber_aux(2)
if(uintSizeInBytes <= 1) then vv |= ber_aux(1)
```

Static Primitives: LOC Statistics

BitStream

- Total LOC: **3700 lines** (proof + implementation)
- Ratio: ~ 5:1

ACN Codec

- Total LOC: **4000 lines** (proof + implementation)

Verification: Generated Code

Tailoring of the Compiler for Verification

Verification: Generated Code

Same high-level idea than for static code

However: **Generate proof automatically**

→ **Tailor the generated code to verification**

Tailoring: Translation to Functional Code

ASN1SCC existing backends (C, Ada) use **in-place mutation for decoding**

Incompatible with the **aliasing policy** of Stainless

Solution: **Functional code**

- Return decoded values
- How to treat SEQUENCE OF?
  - Return a new array?
  - Append decoded elements to a collection?

Tailoring: Replace Arrays by Vectors

Problem: **SEQUENCE OF within SEQUENCE OF**

- **→ Arrays within arrays**
- Incompatible with aliasing policy

Solution: **Replace Arrays by a wrapped Scala Vector**

- **Immutable**
- Append/prepend/init/tail: **O(1)** amortised, worst O(log n)
- Random accesses: **O(log n) → Acceptable**
- **Specified with List for verification**

Verification: Generated Code

Step 1: runtime safety

- Same approach as for static primitives
- Unclear how many bits to read → generate *size* functions

```
MyChoice ::= CHOICE
{
  choice1 SEQUENCE {
    fst INTEGER,
    snd INTEGER
  },
  choice2 INTEGER
}
END
```

```
def size(): Long = {
  this match {
    case TMyChoice.choice1_PRESENT(choice1) =>
      val size_1_0 = 8L * GetLengthForEncodingSigned(choice1.fst) + 8L
      val size_1_1 = 8L * GetLengthForEncodingSigned(choice1.snd) + 8L
      size_1_0 + size_1_1
    case TMyChoice.choice2_PRESENT(choice2) =>
      8L * GetLengthForEncodingSigned(choice2) + 8L
  }
}.ensuring { (res: Long) => (0L <= res) && (res <= 145L)}
```

Verification: Generated Code

Step 2: semantic correctness

- Same approach as for the static code
  - **Relies heavily on BitStream and Codec proven properties**
- **Generated** proof
  - Generate **specifications**
  - Generate **lemmas**
  - Generate **lemma applications**
- Verified **automatically**

See paper for more details

Verified Properties and Statistics

Experimental Results

Packet Formats

Test with real-world packet formats

- PUS-C Services (Packet Utilisation Standard C)
  - **312 packet formats**
  - **Standard packet specification used by ESA** satellites and ground control stations ("ECSS-E-ST-70-41C")
  - Used by e.g. **CHEOPS** (exo-planets transits observation) and **Proba-3** (demonstration of satellites formation flight) missions
- TC-Packet
  - **Telecommand packet format for satellites used by ESA**

Verification conditions statistics

335,149 VCs total

Largest verification project with Stainless to date

	Library			PUS-C services			TC-Packet		
VCs	# V	# U	# I	# V	# U	# I	# V	# U	# I
Preconditions	4,252	0	0	152,201	1	2	529	0	0
Overflows/casts	936	0	0	82,037	0	0	230	0	0
Assertions	544	0	0	23,284	0	0	167	0	0
Postcondition	443	0	0	22,365	1	0	30	0	0
Arithmetic ops	183	0	0	3,711	0	0	0	0	0
Array access	181	0	0	0	0	0	0	0	0
Measures	132	0	0	2,796	0	0	0	0	0
Class invariant	54	0	0	1,722	0	0	0	0	0
Match exh.	39	0	0	38,283	0	0	101	0	0
Pos. array size	5	0	0	0	0	0	0	0	0
Miscellaneous	2	0	0	918	0	0	0	0	0
Total	**6,771**	**0**	**0**	**327,317**	**2**	**2**	**1,057**	**0**	**0**

Lessons

Bugs
Takeaway

Bugs found

1. Incorrect treatment of NaN in C and Ada backends
   - Failing assertions for NaN bit pattern
   - **Found during translation C → Scala**
2. SEQUENCE with alignment requirement
   - Wrong bit paddings
   - **Found during translation C → Scala**
3. Erroneous decoding of some CHOICE pattern
   - When optional and specified with ACN codec
   - **Found while writing proof**
4. 7-bit Strings missing validation
   - Missing range checks for 7-bit strings represented as 8-bit constrained in [0, 127]
   - **Found with Stainless**

→ All bugs are reproducible with Stainless

→ All those bugs are now fixed in all backends

Every step counts

Lots of refactoring needed to verify existing code

- Comply with tool, simplify reasoning about algorithms, …
- → **Led to discovering bugs**

Stainless generates these VCs automatically

- Termination, absence of overflow, in-bounds accesses, …

→ Some verification **without writing specifications**

Every step of the process provides some valuable guarantees

Conclusion

New **verified Scala backend** for ASN1SCC compiler used by ESA missions

- **Static runtime library subset used by PUS-C services verified**
  - Crash free
  - Invertible (except for real numbers and strings operations)
- **Generated proof with the generated Scala code**
  - Generated code verified to be
    - Crash free
    - Invertible for SEQUENCE encoding

→ **Better reliability of Space communications**

samuel.chassot@epfl.ch

Backup slides

Verification conditions statistics

- 2 invalid VCs
  - Precondition checks in IA5String encoding
  - Size of null-terminating strings
  - Adding a check for Scala backend would impact Ada and C backend

	Library			PUS-C services			TC-Packet		
VCs	# V	# U	# I	# V	# U	# I	# V	# U	# I
Preconditions	4,252	0	0	152,201	1	2	529	0	0
Overflows/casts	936	0	0	82,037	0	0	230	0	0
Assertions	544	0	0	23,284	0	0	167	0	0
Postcondition	443	0	0	22,365	1	0	30	0	0
Arithmetic ops	183	0	0	3,711	0	0	0	0	0
Array access	181	0	0	0	0	0	0	0	0
Measures	132	0	0	2,796	0	0	0	0	0
Class invariant	54	0	0	1,722	0	0	0	0	0
Match exh.	39	0	0	38,283	0	0	101	0	0
Pos. array size	5	0	0	0	0	0	0	0	0
Miscellaneous	2	0	0	918	0	0	0	0	0
Total	**6,771**	**0**	**0**	**327,317**	**2**	**2**	**1,057**	**0**	**0**

Verification conditions statistics

- 2 timeout VCs
  - Same nature except Stainless could not find a counterexample

	Library			PUS-C services			TC-Packet		
VCs	# V	# U	# I	# V	# U	# I	# V	# U	# I
Preconditions	4,252	0	0	152,201	1	2	529	0	0
Overflows/casts	936	0	0	82,037	0	0	230	0	0
Assertions	544	0	0	23,284	0	0	167	0	0
Postcondition	443	0	0	22,365	1	0	30	0	0
Arithmetic ops	183	0	0	3,711	0	0	0	0	0
Array access	181	0	0	0	0	0	0	0	0
Measures	132	0	0	2,796	0	0	0	0	0
Class invariant	54	0	0	1,722	0	0	0	0	0
Match exh.	39	0	0	38,283	0	0	101	0	0
Pos. array size	5	0	0	0	0	0	0	0	0
Miscellaneous	2	0	0	918	0	0	0	0	0
Total	**6,771**	**0**	**0**	**327,317**	**2**	**2**	**1,057**	**0**	**0**

The Value of Refactoring

Lots of refactoring needed to verify existing code

- Comply with tool, simplify reasoning about algorithms, …

Stainless generates VCs automatically

- Termination, absence of overflow, in-bounds accesses, …

→Some verification **without writing specifications**

Can make verification a lot easier like with the loop unrolling trick

- Sometimes **easier to rewrite than verify existing**