Towards Verifying the Bitcoin-S Library

Ramon Boss, Kai Brünnler, Anna Doukmak

Bern University of Applied Sciences

July 16, 2020

Table of Contents

1. Bitcoin-S
2. Stainless
3. Rewriting Bitcoin-S for Stainless
4. Bugs We Found

Table of Contents

1. Bitcoin-S
2. Stainless
3. Rewriting Bitcoin-S for Stainless
4. Bugs We Found

Bitcoin-S

```
1     val privKey = ECPrivateKey.freshPrivateKey
2     val spk = P2PKHScriptPubKey(
3       pubKey = privKey.publicKey
4     )
5
6     val amount = Satoshis(Int64(10000))
7
8     val utxo = TransactionOutput(
9       currencyUnit = amount,
10      scriptPubKey = spk
11    )
12
13    val tx = BaseTransaction(
14      version = Int32.one,
15      inputs = List.empty,
16      outputs = List(utxo),
17      lockTime = UInt32.zero
18    )
```

Table of Contents

1. Bitcoin-S
2. Stainless
3. Rewriting Bitcoin-S for Stainless
4. Bugs We Found

Stainless: Example

```
1    def factorial(n: Int): Int = {
2        require(n >= 0)
3        if (n == 0) {
4          1
5        } else {
6          n * factorial(n - 1)
7        }
8    } ensuring(res => res >= 0)
```

Stainless: Example

Stainless output for the factorial function

```
[  Info  ]  - Now solving 'postcondition' VC for factorial @10:3...
[  Info  ]  - Result for 'postcondition' VC for factorial @10:3:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   n: Int -> 17
[  Info  ]   ┌─────────────────┐
[  Info  ] ╔═╡ stainless summary ╞═══════════════════════════════════════════════════════════════════════════════════════════════════════╗
[  Info  ] ╟─└─────────────────┘─────────────────────────────────────────────────────────────────────────────────────────────────────────╢
[  Info  ] ╟ factorial  postcondition                    valid from cache            src/TestFactorial.scala:10:3   1.055 ╢
[  Info  ] ╟ factorial  postcondition                    invalid           U:smt-z3  src/TestFactorial.scala:10:3   7.861 ╢
[  Info  ] ╟ factorial  precond. (call factorial(n - 1)) valid from cache            src/TestFactorial.scala:15:11  1.054 ╢
[  Info  ] ╠═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════╣
[  Info  ] ╟ total: 3    valid: 2    (2 from cache) invalid: 1    unknown: 0    time:   9.970                                       ╢
[  Info  ] ╚═══════════════════════════════════════════════════════════════════════════════════════════════════════════════════════════╝
```

Stainless

Pure Scala

- Stainless works on a subset of Scala called *Pure Scala*
- It's essentially algebraic datatypes (case classes) and pure functions
- Some things that it does not include:
  - inheritance by objects
  - abstract type members
  - inner classes in case objects
  - ...

Table of Contents

1. Bitcoin-S
2. Stainless
3. Rewriting Bitcoin-S for Stainless
4. Bugs We Found

Inheriting Objects

This:

```
1  object Satoshis extends BaseNumbers[Satoshis] {
2    val zero = Satoshis(Int64.zero)
3    val one = Satoshis(Int64.one)
4  }
```

Becomes this:

```
1  case object Satoshis extends BaseNumbers[Satoshis] {
2    val zero = Satoshis(Int64.zero)
3    val one = Satoshis(Int64.one)
4  }
```

Abstract Type Members

This:

```
1  sealed abstract class CurrencyUnit {
2    type A
3
4    protected def underlying: A
5  }
6
7  sealed abstract class Satoshis extends CurrencyUnit {
8    override type A = Int64
9  }
```

Becomes this:

```
1  sealed abstract class CurrencyUnit {
2    protected def underlying: Int64
3  }
4
5  sealed abstract class Satoshis extends CurrencyUnit
```

Missing Bitwise &-Function on BigInt

This:

```
1  sealed abstract class Number {
2    def andMask: BigInt
3    def checkResult(result: BigInt): BigInt = {
4      require((result & andMask) == result)
5      result
6    }
7  }
```

Becomes this:

```
1  sealed abstract class Number {
2    // removed redundant checkResult function
3  }
```

Specification and Stainless Output

```scala
def +(c: CurrencyUnit): CurrencyUnit = {
  require(c.satoshis == Satoshis.zero)
  Satoshis(
    satoshis.underlying + c.satoshis.underlying
  )
} ensuring(res => res.satoshis == this.satoshis)
```

```
[  Info  ]  - Now solving 'postcondition' VC for + @9:3...
[  Info  ]  - Result for 'postcondition' VC for + @9:3:
[  Info  ]  => VALID
[  Info  ]
[  Info  ]  stainless summary
[  Info  ]
[  Info  ]  +  postcondition   valid  U:smt-z3   verified/currency/CurrencyUnits.scala:9:3      1.451
[  Info  ]  ---------------------------------------------------------------------------------------------
[  Info  ]  total: 1    valid: 1    (0 from cache) invalid: 0    unknown: 0    time:   1.451
[  Info  ]
```

Table of Contents

1. Bitcoin-S
2. Stainless
3. Rewriting Bitcoin-S for Stainless
4. Bugs We Found

Bug 1: Function checkTransaction is too restrictive

4 core/src/main/scala/org/bitcoins/core/script/interpreter/ScriptInterpreter.scala

```
@@ -779,8 +779,8 @@ sealed abstract class ScriptInterpreter {
779 779         val totalSpentByOutputs: CurrencyUnit =
780 780           outputValues.fold(CurrencyUnits.zero)(_ + _)
781 781         val allOutputsValidMoneyRange = validMoneyRange(totalSpentByOutputs)
782     -       val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
783     -       val noDuplicateInputs = prevOutputTxIds.distinct.size == prevOutputTxIds.size
    782 +       val prevOutputs = transaction.inputs.map(_.previousOutput)
    783 +       val noDuplicateInputs = prevOutputs.distinct.size == prevOutputs.size
784 784
785 785         val isValidScriptSigForCoinbaseTx = transaction.isCoinbase match {
786 786           case true =>
```

Bug 2: Function checkResult is buggy (and redundant)

22 core/src/main/scala/org/bitcoins/core/number/NumberType.scala

```
@@ -37,10 +37,10 @@ sealed abstract class Number[T <: Number[T]]
37 37     /** Factory function to create the underlying T, for instance a UInt32 */
38 38     def apply: A => T
39 39
40    -   override def +(num: T): T = apply(checkResult(underlying + num.underlying))
41    -   override def -(num: T): T = apply(checkResult(underlying - num.underlying))
42    -   override def *(factor: BigInt): T = apply(checkResult(underlying * factor))
43    -   override def *(num: T): T = apply(checkResult(underlying * num.underlying))
   40 +   override def +(num: T): T = apply(underlying + num.underlying)
   41 +   override def -(num: T): T = apply(underlying - num.underlying)
   42 +   override def *(factor: BigInt): T = apply(underlying * factor)
   43 +   override def *(num: T): T = apply(underlying * num.underlying)
44 44
45 45     override def compare(num: T): Int = underlying compare num.underlying
46 46
@@ -64,20 +64,10 @@ sealed abstract class Number[T <: Number[T]]
64 64       }
65 65     }
66 66
67    -   def |(num: T): T = apply(checkResult(underlying | num.underlying))
68    -   def &(num: T): T = apply(checkResult(underlying & num.underlying))
   67 +   def |(num: T): T = apply(underlying | num.underlying)
   68 +   def &(num: T): T = apply(underlying & num.underlying)
69 69     def unary_- : T = apply(-underlying)
70 70
71    -   /**
72    -     * Checks if the given result is within the range
73    -     * of this number type
74    -     */
75    -   private def checkResult(result: BigInt): A = {
76    -     require((result & andMask) == result,
77    -             "Result was out of bounds, got: " + result)
78    -     result
79    -   }
80    -
```

Conclusion

Lessons learned

- Even verifying very simple properties requires significantly rewriting existing Scala code
- This is true even for purely functional code
- We can't really verify existing code – really we verify a re-implementation of it
- But attempting to verify has been very useful for finding bugs