Towards Verifying the Bitcoin-S Library

Ramon Boss, Kai Brünnler, and Anna Doukmak

Bern University of Applied Sciences, CH-2501 Biel, Switzerland
ramon.boss@outlook.com, kai.bruennler@bfh.ch,
anna.doukmak@gmail.com

Abstract. We try to verify properties of the bitcoin-s library, a Scala implementation of parts of the Bitcoin protocol. We use the Stainless verifier which supports programs in a fragment of Scala called *Pure Scala*. Since bitcoin-s is not written in this fragment, we extract the relevant code from it and perform a series of equivalent transformations until we arrive at code that we successfully verify. In that process we find and fix two bugs in bitcoin-s.

Keywords: Bitcoin · Scala · bitcoin-s · Stainless.

1 Introduction

For software handling cryptocurrency, correctness is clearly crucial. However, even in very well-tested software such as Bitcoin Core, serious bugs occur. The most recent example is the bug found in September 2018 [10] which essentially allowed to arbitrarily create new coins. Such software is thus a worthwhile target for formal verification. In this work, we set out to verify properties of the bitcoin-s library with the Stainless verifier.

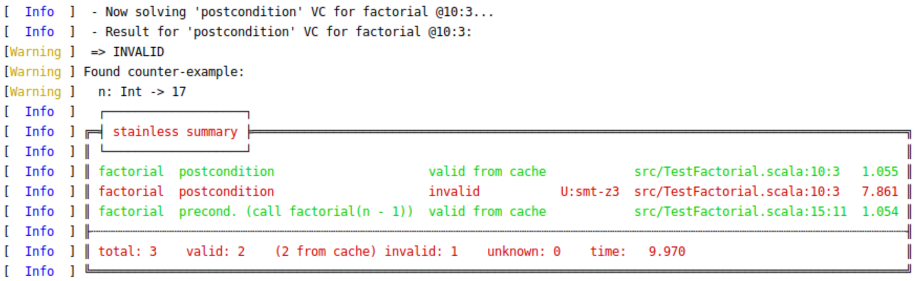
The Bitcoin-S Library. The bitcoin-s library is an implementation of parts of the Bitcoin protocol in Scala [11,12]. In particular, it allows to serialize, deserialize, sign and validate Bitcoin transactions. The library uses immutable data structures and algebraic data types but is not specifically written with formal verification in mind. According to the website, the library is used in production, handling significant amounts of cryptocurrency each day [11].

The Stainless Verifier. Stainless is the successor of the Leon verifier and is developed at EPF Lausanne [2,14,1]. It is intended to be used by programmers without training in formal verification. To facilitate that, it accepts specifications written in the programming language itself (Scala). Also, it focusses on counterexample finding in addition to proving correctness. Counterexamples are useful to programmers while correctness proofs are not – correctness is obvious or does not hold, and often both at the same time.

The example in Figure 1 adapted from the Stainless documentation [8] shows how the verifier is used. Note how a precondition is specified using `require` and a postcondition using `ensuring`. Our function does not satisfy the specification. An overflow in the 32-bit integer type leads to a negative result for the input 17,

```scala
def factorial(n: Int): Int = {
    require(n >= 0)
    if (n == 0) {
      1
    } else {
      n * factorial(n - 1)
    }
} ensuring(res => res >= 0)
```

Fig. 1. Factorial function with specification



```
[  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  ] ╚═══════════════════════════════════════════════════════════════════════════════════════════════════════════════╝
```

Fig. 2. Stainless output for the factorial function

as Stainless reports in Figure 2. Changing the type `Int` to `BigInt` will result in a successful verification.

The Pure Scala Fragment. The Scala fragment supported by Stainless comprises algebraic data types in the form of abstract classes, case classes and case objects, objects for grouping classes and functions, boolean expressions with short-circuit interpretation, generics with invariant type parameters, pattern matching, local and anonymous classes and more. In addition to Pure Scala Stainless also supports some imperative features, such as using a (mutable) variable in a local scope of a function and while loops. They turn out not to be relevant for our current work.

What will turn out to be more relevant for us are the Scala features which Stainless does not support, such as: inheritance by objects, abstract type members, and inner classes in case objects. Also, Stainless has its own library of some core data types and functions which are mapped to corresponding data types and functions inside of the SMT solver that Stainless ultimately relies on. Those data types in general do not have all the methods of the Scala data types. For example, the `BigInt` type in Scala has methods for bitwise operations while the `BigInt` type in Stainless does not.

Outline and Properties to Verify. In the next section we try to verify the property that a regular (non-coinbase) transaction can not generate new coins. We call it the *no-inflation property*. Trying to verify it, we uncover and fix a bug in the bitcoin-s library. We then find that there is too much code involved that lies outside of the supported fragment to currently make this verification feasible. So we turn to a simpler property to verify. The simplest possible property we can think of is the fact that adding zero satoshis to a given amount of satoshis yields the given amount of satoshis. We call it the *addition-with-zero property* and we try to verify it in Section 3. Here as well we see that a significant part of the code lies outside of the supported fragment. We perform a series of equivalent transformations on it until we arrive at code that we successfully verify. In that process we find and fix a second bug in bitcoin-s.

2 The No-Inflation Property

The `checkTransaction` function shown in Figure 3 is crucial for the verification of the no-inflation property. Given a transaction it returns true if some basic checks succeed, otherwise false. For example, one of those checks is that both the list of inputs and list of outputs need to be non-empty.

To better understand the validation of a transaction in bitcoin-s, it is useful to review how transactions are represented and created.

Creating a Transaction. To create a transaction, we first need some coins – an unspent transaction output. We could load an actual unspent transaction output from the bitcoin network, but we create one manually in order to see this process. So we first create an (invalid) transaction with one output in Figure 4.

We first create a keypair, then a lock script with its public key, then the amount of satoshis, then a transaction output (`utxo`) for that amount and locked with that script. Finally we create a transaction with that output and no inputs. Of course, that is not a valid transaction, because it creates coins out of nothing. In particular, `checkTransaction(prevTx)` returns false, simply because the list of inputs is empty.

Now that we have a transaction output, we create a transaction to spend it in Figure 5. First, we need a reference to an output of a previous transaction, here called `outPoint`. Second, we add some information on how to spend that output, in particular, how to sign the transaction. Now we assemble the list of unspent transaction outputs (`utxos`), in our case just one.

We then set the amount of satoshis that we want to spend. The `Int64` class aims to emulate a C data type in Scala, and we will look at it more closely in the next section.

We then create a lock script (`destinationSPK`) to receive the coins, create our list of transaction outputs (`destinations`), define the fee rate and set some bitcoin network parameters.

Now we create a transaction builder with those data and we tell it to start signing the transaction in line 34.

Finally, we get the actual signed transaction. We could serialize it and send it to the Bitcoin network. We can also pass it to the checkTransaction function, which will return true.

A Bug in the checkTransaction Function. Note lines 15-17 in Figure 3. Here, the value `prevOutputTxIds` gathers a list of all transaction identifiers referenced

```
def checkTransaction(transaction: Transaction): Boolean = {
  val inputOutputsNotZero =
    !(transaction.inputs.isEmpty || transaction.outputs.isEmpty)
  val txNotLargerThanBlock =
    transaction.bytes.size < Consensus.maxBlockSize
  val outputsSpendValidAmountsOfMoney =
    !transaction.outputs.exists(o =>
      o.value < CurrencyUnits.zero || o.value > Consensus.maxMoney)

  val outputValues = transaction.outputs.map(_.value)
  val totalSpentByOutputs: CurrencyUnit =
    outputValues.fold(CurrencyUnits.zero)(_ + _)
  val allOutputsValidMoneyRange =
    validMoneyRange(totalSpentByOutputs)
  val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
  val noDuplicateInputs =
    prevOutputTxIds.distinct.size == prevOutputTxIds.size

  val isValidScriptSigForCoinbaseTx = transaction.isCoinbase match {
    case true =>
      transaction.inputs.head.scriptSignature.asmBytes.size >= 2 &&
        transaction.inputs.head.scriptSignature.asmBytes.size <= 100
    case false =>
      !transaction.inputs.exists(
        _.previousOutput == EmptyTransactionOutPoint)
  }
  inputOutputsNotZero && txNotLargerThanBlock &&
  outputsSpendValidAmountsOfMoney && noDuplicateInputs &&
  allOutputsValidMoneyRange && noDuplicateInputs &&
  isValidScriptSigForCoinbaseTx
}
```

Fig. 3. The `checkTransaction` function

```
val privKey = ECPrivateKey.freshPrivateKey
val creditingSPK = P2PKHScriptPubKey(pubKey = privKey.publicKey)

val amount = Satoshis(Int64(10000))

val utxo = TransactionOutput(currencyUnit = amount, scriptPubKey =
    creditingSPK)

val prevTx = BaseTransaction(
  version = Int32.one,
  inputs = List.empty,
  outputs = List(utxo),
  lockTime = UInt32.zero
)
```

Fig. 4. Creating a transaction output to spend

by the inputs of the current transaction. If the size of this list is the same as the size of this list with duplicates removed, we know that no transaction has been referenced twice. This prevents a transaction from spending two different outputs of the same previous transaction. The check is too strict: `checkTransaction` returns false for valid transactions.

The fix is simple: we perform the duplicate check on the `TransactionOutPoint` instances instead of on their transaction identifiers. Note that `TransactionOutPoint` is a case class and thus its notion of equality is just what we need: equality of of both the transaction identifier and the output index.

Specifically, we replace lines 15-17 as follows:

```
15  val prevOutputs = transaction.inputs.map(_.previousOutput)
16  val noDuplicateInputs =
17    prevOutputs.distinct.size == prevOutputs.size
```

We submitted this fix together with a corresponding unit test to the bitcoin-s project in a pull request, which has been merged [5].

An Attempt at Verification. Naively trying Stainless on the entire bitcoin-s codebase results in many errors – as was to be expected. We tried to extract only the code relevant to the no-inflation-property and to verify that. However, even the extracted code has more than 1500 lines and liberally uses Scala features outside of the supported fragment. We tried to transform the code into the supported fragment, but quickly realized that a better approach is to first verify a simpler property depending on less code and later come back to the no-inflation property with more experience. So we now turn to the addition-with-zero property.

```
val outPoint = TransactionOutPoint(prevTx.txId, UInt32.zero)

val utxoSpendingInfo = BitcoinUTXOSpendingInfo(
  outPoint = outPoint,
  output = utxo,
  signers = List(privKey),
  redeemScriptOpt = None,
  scriptWitnessOpt = None,
  hashType = HashType.sigHashAll
)

val utxos = List(utxoSpendingInfo)

val destinationAmount = Satoshis(Int64(5000))

val destinationSPK = P2PKHScriptPubKey(pubKey = ECPrivateKey.
    freshPrivateKey.publicKey)

val destinations = List(
  TransactionOutput(currencyUnit = destinationAmount, scriptPubKey
      = destinationSPK)
)

val feeRate = SatoshisPerByte(Satoshis.one)

val networkParams = RegTest // some static values for testing

val txBuilderF: Future[BitcoinTxBuilder] = BitcoinTxBuilder(
  destinations = destinations,
  utxos = utxos,
  feeRate = feeRate,
  changeSPK = creditingSPK,    // where to send the change
  network = networkParams
)

val txF: Future[Transaction] = txBuilderF.flatMap(_.sign)

val tx: Transaction = Await.result(txF, 1 second)
```

Fig. 5. Creating a transaction

3 The Addition-with-Zero Property

It is of course a crucial property we are verifying here: if zero satoshis were credited to your account, you would not want your balance to change! It is also the simplest meaningful property to verify that we can think of. However, the code involved in performing the addition of two satoshi amounts in bitcoin-s is nontrivial. The reason for that is a peculiarity of consensus code: agreement with the majority is more important than correctness, whatever correctness might mean. The most widely used bitcoin implementation by far is the reference implementation Bitcoin Core, written in C++. For consensus code, bitcoin-s has little choice but to be in strict agreement with the reference implementation. To achieve that, it implements C-like data types in Scala and then implements functionality using those C-like data types. For example, the Satoshis class, which represents an amount of satoshis, is implemented using the class `Int64` which aims to represent the C-type `int64_t`.

Extracting the Relevant Code. The relevant code for the addition of satoshis is in two files: CurrencyUnits.scala and NumberType.scala. From those files we removed the majority of the code because it is not needed for the verification of our property. For example, we removed all number types except for `Int64` (so `Int32`, `UInt64`, etc.) because they are not used. We also removed the superclasses `Factory` and `NetworkElement` of `CurrencyUnit` and `Number`, respectively, because the inherited members are not used. We further removed all binary operations on `Number` that are not used, like subtraction and multiplication. The extracted code is shown in Figure 6 and Figure 7.

A Bug in the checkResult Function. Note the `checkResult` function on line 12 and the value `andMask` on line 23 of NumberType.scala. The function is intended to catch overflows by performing a bitwise conjunction of its argument with `andMask` and comparing the result with the argument. However, because of the way Java BigIntegers are represented [15] and because bitwise operations implicitly perform a sign extension [9] on the shorter operand, the function does not actually catch overflows.

While this is a potentially serious bug, it turns out that `checkResult` is only ever called inside a constructor call for a number type which contains the intended range check, see lines 32-35. The `checkResult` function thus can, and should, be removed entirely. The bitcoin-s developers have acknowledged the bug and we submitted a pull request to fix it [4].

Transforming the Code. We now turn to the list of Scala features used by the extracted code which are not supported by Stainless and how to transform the code into the supported fragment. All transformations are *equivalent* in the sense that if the addition-with-zero property holds for the transformed code, then it also holds for the code before the transformation.

Inheriting Objects. In both files we have objects extending the BaseNumbers trait, on lines 30 and 23 respectively, which Stainless does not support. We simply turn those objects into case objects. That transformation is equivalent: case objects have various additional properties (for example, being serializable) but none of our code depends on the absence of those.

```
package extracted.number

sealed abstract class Number[T <: Number[T]] {
  type A = BigInt
  protected def underlying: A
  def toLong: Long = toBigInt.bigInteger.longValueExact()
  def toBigInt: BigInt = underlying
  def andMask: BigInt
  def apply: A => T
  def +(num: T): T = apply(checkResult(underlying + num.underlying))

  private def checkResult(result: BigInt): A = {
    require((result & andMask) == result,
      "Result␣was␣out␣of␣bounds,␣got:␣" + result)
    result
  }
}

sealed abstract class SignedNumber[T <: Number[T]] extends Number[T]

sealed abstract class Int64 extends SignedNumber[Int64] {
  override def apply: A => Int64 = Int64(_)
  override def andMask = 0xffffffffffffffffL
}

trait BaseNumbers[T] {
  def zero: T
}

object Int64 extends BaseNumbers[Int64] {
  private case class Int64Impl(underlying: BigInt) extends Int64 {
    require(underlying >= -9223372036854775808L,
      "Number␣was␣too␣small␣for␣a␣int64,␣got:␣" + underlying)
    require(underlying <= 9223372036854775807L,
      "Number␣was␣too␣big␣for␣a␣int64,␣got:␣" + underlying)
  }

  lazy val zero = Int64(0)
  def apply(long: Long): Int64 = Int64(BigInt(long))
  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

Fig. 6. Extracted Code from NumberType.scala

```scala
package extracted.currency

import extracted.number.{BaseNumbers, Int64}

sealed abstract class CurrencyUnit {
  type A
  def satoshis: Satoshis
  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
  def +(c: CurrencyUnit): CurrencyUnit = {
    Satoshis(satoshis.underlying + c.satoshis.underlying)
  }
  protected def underlying: A
}

sealed abstract class Satoshis extends CurrencyUnit {
  override type A = Int64
  override def satoshis: Satoshis = this
  def toBigInt: BigInt = BigInt(toLong)
  def toLong: Long = underlying.toLong
  def ==(satoshis: Satoshis): Boolean = underlying == satoshis.underlying
}

object Satoshis extends BaseNumbers[Satoshis] {
  val zero = Satoshis(Int64.zero)
  def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
  private case class SatoshisImpl(underlying: Int64) extends Satoshis
}
```

Fig. 7. Extracted Code from CurrencyUnits.scala

Abstract Type Members. In CurrencyUnits.scala on line 6 there is an abstract type that is not supported. Note that we can not simply replace it with a (supported) type parameter since the CurrencyUnit class uses one of its implementing classes: Satoshis. Since the Satoshis class overrides `A` with `Int64` anyway, we just remove the abstract type declaration and replace `A` by `Int64` everywhere.

Non-Literal BigInt Constructor Argument. In CurrencyUnits.scala on line 18 the BigInt constructor is called with a non-literal argument. As described before, the types in the Stainless library are more restricted than their Scala library counterparts. In particular, the Stainless BigInt constructor is restricted to literal arguments. So we simply replace `toLong` by `underlying.toBigInt`: instead of converting the underlying `Int64` (which in turn has an underlying `BigInt`) to `Long` and then back to `BigInt` we simply directly return the `BigInt`. This is an equivalent transformation: the only thing that might go wrong in the detour via `Long` is that the underlying `BigInt` does not fit into a `Long`. However, the only constructor of `Int64Impl` ensures exactly that and all functions producing `Int64` do so via this constructor.

Self-Reference in Type Parameter Bound. In NumberTypes.scala both on lines 3 and 19 is a class with a type parameter and a type boundary that contains that type parameter itself. Stainless does not currently support such self-referential type boundaries. We opened an issue [3] on the Stainless repository and the developers have targeted version 0.4 to support self-referential type boundaries. Since our code only uses Number with type parameter `T` instantiated to `Int64`, we just remove the type parameter declaration and replace all its occurrences by `Int64`.

Missing Member `bigInteger` in BigInt. In NumberType on line 6 there is a reference to `bigInteger`. The Scala `BigInt` class is essentially a wrapper around `java.math.BigInteger`. `BigInt` has a member `bigInteger` which is the underlying instance of the Java class. The Java class has a method `longValueExact` which returns a `long` only if the `BigInteger` fits into a `long`, otherwise throws exception. Stainless does not support Java classes and in particular its `BigInt` has no member `bigInteger`. However, our code does not call `toLong` anymore, so we just remove it.

Type Members. In NumberType.scala there is a type member on line 4. Our version of Stainless (0.1) does not support type members. We just remove the declaration and replace all occurrences of `A` with `BigInt`, since `A` is never overwritten in an implementing class. Note that in the meantime Stainless has implemented support for type members [13]. Since version 0.2 verification should succeed without this change.

Missing Bitwise-And Method on BigInt. Contrary to Scala `BigInt`, the Stainless `BigInt` class does not support bitwise operations, in particular not the &-method used in NumberType.scala on line 13. However, as described above, the `checkResult` function is both broken and redundant, so we remove it and all calls to it.

Inner Class in Case Object. We have inner classes in NumberType.scala on line 31 and in CurrencyUnits.scala on line 26. Stainless does not support inner classes in a case object. We just move the inner classes out of the case objects. They do not interfere with any other code.

Message Parameter in Require. The calls of the require function on lines 32 and 34 in CurrencyUnits.scala have a second parameter: the error message. Stainless does not support the message parameter. We simply remove it.

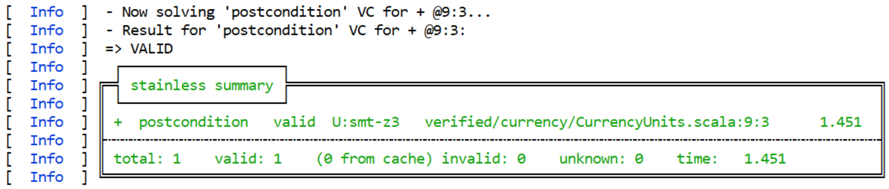
Missing Implicit Long to BigInt Conversion. The Scala `BigInt` class has implict conversions from `Long` which NumberType.scala uses on lines 32 and 34. They are missing in the Stainless `BigInt`. A `BigInt` constructor with a `Long` argument is also missing. We thus replace the `Long` literals by an explicit call to the `BigInt` constructor with a literal string argument, e.g. `BigInt("-9223...5808")`.

The Specification. Now that all our code has been transformed into the supported fragment, we can finally write our specification, shown in Figure 8, and verify it with Stainless, as the output in Figure 9 shows.

The original bitcoin-s code we started from, the extracted code, and the finally verified code are available in our GitHub repository [6].

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

Fig. 8. Addition function with specification



```
[  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  ]
```

Fig. 9. Stainless output for the transformed code

4 Conclusion and Future Work

We are happy to see some friendly green verifier output. However, apart from the bugs we found, the main conclusion from this work is that we had to non-trivially transform even a very small portion of the code in order to verify it. At the moment, it is unrealistic to routinely formally verify properties as part of the bitcoin-s development process. However, Stainless development has already progressed (e.g. type members are supported in recent versions) and continues to do so (e.g. self-referential type bounds are on the roadmap). Some missing features that we identified are presumably very easy to support, like the message parameter in the require function. Some other features presumably require more substantial work, like bitwise operations on integer types.

On the other hand, bitcoin-s uses features that might not be supported even by future Stainless versions, such as calls to Java code. Here the bitcoin-s code can hopefully be adapted to accommodate formal verification.

From our results we conclude that formal verification of bitcoin libraries in general and bitcoin-s in particular is a worthwhile endeavour. We are looking forward to verifying more substantial parts of the code in future work.

References

1. Blanc, R., Kuncak, V.: Sound reasoning about integral data types with a reusable SMT solver interface. In: Haller and Miller [7], pp. 35–40. https://doi.org/10.1145/2774975.2774980
2. Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013. pp. 1:1–1:10. ACM (2013). https://doi.org/10.1145/2489837.2489838
3. Boss, R.: Issue 519: Unknown type parameter type T in self referencing generic, https://github.com/epfl-lara/stainless/issues/519, accessed 2019-06-27
4. Boss, R.: Remove redundant function checkresult, https://github.com/bitcoin-s/bitcoin-s/pull/565, accessed 2019-07-03
5. Boss, R.: Transaction can reference two different outputs of the same previous transaction, https://github.com/bitcoin-s/bitcoin-s/pull/435, accessed 2019-06-19
6. Boss, R., Brünnler, K., Doukmak, A.: The bitcoin-s-verification repository, https://github.com/kaibr/bitcoin-s-verification, accessed 2019-07-06
7. Haller, P., Miller, H. (eds.): Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015. ACM (2015)
8. LARA Lab, École Polytechnique Fédérale de Lausanne: Stainless documentation, https://epfl-lara.github.io/stainless/, accessed 2019-06-19
9. Oracle and/or its affiliates: Class BigInteger, https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html, accessed 2019-07-03
10. Song, J.: Bitcoin Core Bug CVE-2018–17144: An Analysis, https://hackernoon.com/bitcoin-core-bug-cve-2018-17144-an-analysis-f80d9d373362, accessed 2019-06-20
11. Suredbits & the bitcoin-s developers: The bitcoin-s website, https://bitcoin-s.org, accessed 2019-06-19
12. The bitcoin-s developers: The bitcoin-s repository, https://github.com/bitcoin-s, accessed 2019-06-19
13. The Stainless developers: Type aliases, type members, and dependent function types, https://github.com/epfl-lara/stainless/pull/470, accessed 2019-06-27
14. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Haller and Miller [7], pp. 18–29. https://doi.org/10.1145/2774975.2774978
15. Wikipedia contributors: Two's complement, https://en.wikipedia.org/wiki/Two%27s_complement, accessed 2019-07-03