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 subset of Scala called the *Pure Scala Fragment*. We first try to verify the property that regular transactions do not create new money. It turns out that there is too much code involved that lies outside of the supported fragment to make this feasible. However, in the process we uncover and fix a bug in bitcoin-s. We then turn to a much simpler (and less interesting) property: that adding zero satoshis to a given amount of satoshis yields the given amount of satoshis. Here as well a significant part of the relevant code lies outside of the supported fragment. However, after a series of equivalent transformations we arrive at code that we successfully verify.

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 [5] 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 [7,8]. In particular, it allows to serialize, deserialize, sign and validate transactions. The library uses immutable data structures and algebraic data types but is not written with formal verification in mind. According to the website, the library is used in production, handling significant amounts of cryptocurrency each day [7].

The Stainless Verifier. Stainless is the successor of the Leon verifier [2,9,1] and is developed at EPF Lausanne [3]. It is intended to be used by programmers without training in formal verification and thus allows to write specifications in Scala and focusses on counterexample finding in addition to proving correctness.

The example in Figure 1 from the Stainless documentation [6] demonstrates this. Notice how a precondition is specified using the function *require* and a postcondition using *ensuring*.

```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 program with specification

Our program happens not to satisfy the specification. An overflow in the 32-bit Int leads to a negative result for the input 17, as Stainless reports in Figure 2. Changing the type from Int to BigInt will result in a successful verification.

```
[Warning ] The Z3 native interface is not available. Falling back onto smt-z3.
[  Info  ]  - Checking cache: 'postcondition' VC for factorial @10:3...
[  Info  ]  - Checking cache: 'precond. (call factorial(n - 1))' VC for factorial @15:11...
[  Info  ]  - Checking cache: 'postcondition' VC for factorial @10:3...
[  Info  ] Cache miss: 'postcondition' VC for factorial @10:3...
[  Info  ] Cache hit: 'precond. (call factorial(n - 1))' VC for factorial @15:11...
[  Info  ] Cache hit: 'postcondition' VC for factorial @10:3...
[  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  ]
[  Info  ] Shutting down executor service.
```

Fig. 2. Output of Stainless verification for calculating factorial of Int number

The Pure Scala Fragment. The Scala fragment supported by Stainless is described in the Stainless documentation [6] in the section Pure Scala.

It 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, default values of function 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 the current work.

What will turn out to be more relevant are the following Scala features which Stainless does not support, such as: (concrete) class definitions, inheritance by objects, abstract type members, variance annotations and private inner classes.

In addition, Stainless has its own library of some core data types and functions which need to be mapped correctly to 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 a 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 cannot 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. Instead, 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. However, after a series of equivalent transformations we arrive at code that we successfully verify.

2 The No-Inflation Property

A crucial function for the verification of the no-inflation property is the `checkTransaction` function, shown in Figure 5. To better understand it, we first see how to create a transaction.

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 3.

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 the actual 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) is false.

Now that we have a transaction output, we create a new transaction to spend it.

First, we need some out points. They point to outputs of previous transactions. We use the index zero, because the previous transaction has only one output that becomes the first index zero. If there were two previous outputs, the second output would become the index 1 and so on.

This utxos are the inputs of our transaction. Second, we need destinations to spend the bitcoins to. For the sake of convenience we create only one. We spend 5000 satoshis to the newly created random public key. Finally, we define the fee rate in satoshis per one byte transaction size as well as some bitcoin network

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

Fig. 3. Creating a transaction output to spend

parameters. The bitcoin network parameters are not important, so we use some static values normally used when testing.

Now lets build the transaction with those data. Line one to seven creates a transaction builder which is then signed on line ten. We can now use our transaction object on line twelve. For example, after calling *hex* on it, we can send the returned string to the bitcoin network.

Validating a Transaction

Bitcoin-S offers a function called *checkTransaction* located in the ScriptInterpreter object.

We can pass a transaction and it returns a Boolean indicating whether the transaction is valid or not. So for example when we pass the transaction we built before the returned value would be true, because it's a valid transaction. It might not be accepted by the bitcoin network but for a transaction on its own it's valid. We can not check context with it, because we can only pass one transaction.

There are several checks in checkTransaction. For example, it checks if there is either no input or no output. In this case we get false.

The relevant part for the bug we found:

```
1  val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
2  val noDuplicateInputs = prevOutputTxIds.distinct.size ==
       prevOutputTxIds.size
```

It gathers all transaction ids referenced by the out points. When we call *distinct* on the returned list, we get a list with duplicate removed. If the size of the new list is the same as the size of the old, we know that there was no duplicate transaction id, because, as said, distinct removes the duplicates.

Trying Stainless on the entire project.

– result sbt: no output

```
1   val outPoint = TransactionOutPoint(prevTx.txId, UInt32.zero)
2
3   val utxoSpendingInfo = BitcoinUTXOSpendingInfo(
4     outPoint = outPoint,
5     output = utxo,
6     signers = List(privKey),
7     redeemScriptOpt = None,
8     scriptWitnessOpt = None,
9     hashType = HashType.sigHashAll
10  )
11
12  val utxos = List(utxoSpendingInfo)
13
14  val destinationAmount = Satoshis(Int64(5000))
15
16  val destinationSPK = P2PKHScriptPubKey(pubKey = ECPrivateKey.
        freshPrivateKey.publicKey)
17
18  val destinations = List(
19    TransactionOutput(currencyUnit = destinationAmount, scriptPubKey
          = destinationSPK)
20  )
21
22  val feeRate = SatoshisPerByte(Satoshis.one)
23
24  val networkParams = RegTest // some static values for testing
25
26  val txBuilderF: Future[BitcoinTxBuilder] = BitcoinTxBuilder(
27    destinations = destinations, // where to send the money
28    utxos = utxos,               // unspent transaction outputs
29    feeRate = feeRate,           // fee rate per byte
30    changeSPK = creditingSPK,    // where to send the change
31    network = networkParams      // bitcoin network information
32  )
33
34  val txF: Future[Transaction] = txBuilderF.flatMap(_.sign)
35
36  val tx: Transaction = Await.result(signedTxF, 1 second)
```

Fig. 4. Creating a transaction

```
/**
  * Checks the validity of a transaction in accordance to bitcoin
      core's CheckTransaction function
  * https://github.com/bitcoin/bitcoin/blob/
      f7a21dae5dbf71d5bc00485215e84e6f2b309d0a/src/main.cpp#L939.
  */
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 =>
      //since this is not a coinbase tx we cannot have any empty
          previous outs inside of inputs
      !transaction.inputs.exists(_.previousOutput ==
          EmptyTransactionOutPoint)
  }
  inputOutputsNotZero && txNotLargerThanBlock &&
      outputsSpendValidAmountsOfMoney && noDuplicateInputs &&
  allOutputsValidMoneyRange && noDuplicateInputs &&
      isValidScriptSigForCoinbaseTx
}
```

Fig. 5. The checkTransaction function in the ScriptInterpreter object

– result jar:

In order to verify a project, Stainless must be integrated into it. We can integrate it in an sbt project adding the Stainless Plugin and the required resolver to pugins.sbt. Another option to use Stainless is to import its libraries in a program code and verify a program from command line using the pre-packaged Stainless JAR file or using the Stainless script built from source. Trying to integrate Stainless in Bitcoin-S caused a lot of troubles, mainly because of version conflicts. For more details see chapter **??**.

Kai: about errors, is it enough to reference to the chapter 4.2 where they are described??

After integrating the Stainless plugin in the Bitcoin-S sbt project, there were many errors because of the different sbt versions. Some errors are described in the section 4.2. It takes too much time to fix them all so it should be easier to extract the classes needed for the `checkTransaction` function.

Putting aside the No-Inflation Property. Naively trying Stainless on the entire bitcoin-s codebase results in either no output (with sbt) or many errors (with jar). So we extract the relevant code to only verify that. However, 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 realize that a better approach is to first verify a simpler property with less code involved and then turn back to the no-inflation property with more experience. So we turn to the addition-with-zero property in the next section.

explain

Fixing a Bug in Bitcoin-S

We can see that there is a bug in the checkTransaction function from before, recognized and fixed through this work.

Here is the relevant code of checkTransaction again:

```
1  val prevOutputTxIds = transaction.inputs.map(_.previousOutput.txId)
2  val noDuplicateInputs = prevOutputTxIds.distinct.size ==
       prevOutputTxIds.size
```

What happens if we have two TransactionOutPoints (previousOutputs) with a different index but referencing the same Transaction ID (txId)?

According to the Bitcoin protocol this is possible. A transaction can have multiple outputs that should be referenceable by the next transaction. So this is clearly a bug.

What should not be possible is a transaction referencing the same output twice. This bug occurred in Bitcoin Core known as CVE-2018–17144 which was patched on September 18, 2018. [5]

Here, Bitcoin-S did a bit too much and marked all transaction as invalid, if they referenced the same transaction twice. The fix is, to check on TransactionOutPoint instead of TransactionOutPoint.txId, because TransactionOutPoint contains the txId as well as the output index it references. So in pseudo code, we check on the tuple (tx, index) instead of (tx). The fixed code:

```
1  val prevOutputs = transaction.inputs.map(_.previousOutput)
```

```
2   val noDuplicateInputs = prevOutputs.distinct.size == prevOutputs.
        size
```

Since TransactionOutPoint is a case class and Scala has a built in == for case classes there is no need to implement TransactionOutPoint.==.

This was fixed in pull request number 435 on GitHub at April 23, 2019, through this work along with a unit test to prevent this bug from appearing again in the future.

3 The Addition-with-Zero Property

In Bitcoin-S there is a class `Satoshis` representing an amount of bitcoins. We look at the verification of the addition of Satoshis with zero Satoshis. This operation should result in the same amount of Satoshis. Let's call it the **??**.

Using Stainless, we see the successful verification of this property. But the process of the verification with the tool requires many changes in the code, so that Stainless can accept it. We look at all needed modifications in chapter 3.

After realizing that it would consume too much time to rewrite the Bitcoin-S code and even the extracted part with checkTransaction, the smallest unit in Bitcoin-S-Core that is worthwhile to verify was extracted. This could be the addition of two CurrencyUnits. To make it even easier, the addition of CurrencyUnits with zero. CurrencyUnits is an abstract class in Bitcoin-S, representing currencies like Satoshis.

Extracting the relevant Code

kai text

extracted 2 files (they are in code/addition/src/main/scala/addition/original)

changes from original to reduced (new folder): - package name

- removed numbers other than Int64

- removed extending Factory, NetworkElement and BasicArithmetic. This includes some hex/byte conversion eg fromHex, hex, bytes, fromBytes, ... Just interfaces never referenced description in section #the-basics (cannot add link with hashtag) NetworkElement class Factory class

- removed Bitcoins class

- removed subtraction and multiplication, binary operations «, », etc, comparision operator <=, >=, etc but not == and !=

- removed toBigDecimal

- removed object CurrencyUnits containing some variables to transform satoshis to btc (not used)

The code we use for the following sections is in reduced folder.

ramon: write this section

ramon: keep close to original code

ramon: for every transformation: before and after

ramon,anna: for every transformation: why does it preserve the semantics

ramon,anna: add comments

ramon,anna: explain what the code does and why

Here we can see the extracted code needed for the addition of CurrencyUnits:

```
1  package addition.reduced.number
2
3  /**
4    * This abstract class is meant to represent a signed and unsigned
       number in C
5    * This is useful for dealing with codebases/protocols that rely on
       C's
6    * unsigned integer types
7    */
8  sealed abstract class Number[T <: Number[T]] {
9    type A = BigInt
10
11   /** The underlying scala number used to to hold the number */
12   protected def underlying: A
13
14   def toLong: Long = toBigInt.bigInteger.longValueExact()
15   def toBigInt: BigInt = underlying
16
17   /**
18     * This is used to determine the valid amount of bytes in a number
19     * for instance a UInt8 has an andMask of 0xff
20     * a UInt32 has an andMask of 0xffffffff
21     */
22   def andMask: BigInt
23
24   /** Factory function to create the underlying T, for instance a
       UInt32 */
25   def apply: A => T
26
27   def +(num: T): T = apply(checkResult(underlying + num.underlying))
28
29   /**
30     * Checks if the given result is within the range
31     * of this number type
32     */
33   private def checkResult(result: BigInt): A = {
34     require((result & andMask) == result,
35       "Result was out of bounds, got: " + result)
36     result
37   }
38 }
39
40 /**
41   * Represents a signed number in our number system
42   * Instances of this is [[Int64]]
43   */
44 sealed abstract class SignedNumber[T <: Number[T]] extends Number[T]
45
```

```
/**
  * Represents a int64_t in C
  */
sealed abstract class Int64 extends SignedNumber[Int64] {
  override def apply: A => Int64 = Int64(_)
  override def andMask = 0xffffffffffffffffL
}

/**
  * Represents various numbers that should be implemented
  * inside of any companion object for a number
  */
trait BaseNumbers[T] {
  def zero: T
  def one: T
  def min: T
  def max: 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)
  lazy val one = Int64(1)

  lazy val min = Int64(-92233720368547758O8L)
  lazy val max = Int64(9223372036854775807L)

  def apply(long: Long): Int64 = Int64(BigInt(long))

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

```
package addition.reduced.currency

import addition.reduced.number.{BaseNumbers, Int64}

sealed abstract class CurrencyUnit {
  type A

  def satoshis: Satoshis

  def !=(c: CurrencyUnit): Boolean = !(this == c)

  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
```

```
13
14    def +(c: CurrencyUnit): CurrencyUnit = {
15      Satoshis(satoshis.underlying + c.satoshis.underlying)
16    }
17
18    protected def underlying: A
19  }
20
21  sealed abstract class Satoshis extends CurrencyUnit {
22    override type A = Int64
23
24    override def satoshis: Satoshis = this
25
26    def toBigInt: BigInt = BigInt(toLong)
27
28    def toLong: Long = underlying.toLong
29
30    def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
          underlying
31  }
32
33  object Satoshis extends BaseNumbers[Satoshis] {
34
35    val min = Satoshis(Int64.min)
36    val max = Satoshis(Int64.max)
37    val zero = Satoshis(Int64.zero)
38    val one = Satoshis(Int64.one)
39
40    def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
41
42    private case class SatoshisImpl(underlying: Int64) extends Satoshis
43  }
```

The additions' signature looks like this:

```
1   +(c: CurrencyUnit): CurrencyUnit
```

When we run Stainless on this code (without any properties to prove), it throws the following errors:

describe errors: no support for abstract types, unsupported arguments for the BigInt constructor, unsupported inheritance for objects.

Transforming the Code

kai todo

Result

Finally, everything is green and correctly verified.

The verified code.

```
1   package addition.modified.number
2
3   /**
```

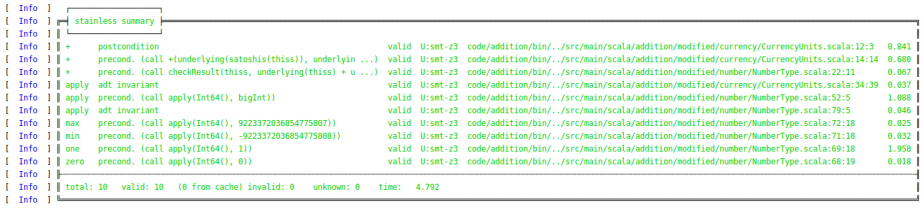

Fig. 6. Output of Stainless verification for addition with 0 of Bitcoin-S-Cores CurrencyUnit

```
4    * This abstract class is meant to represent a signed and unsigned
         number in C
5    * This is useful for dealing with codebases/protocols that rely on
         C's
6    * unsigned integer types
7    */
8  sealed abstract class Number {
9    /** The underlying scala number used to to hold the number */
10   protected def underlying: BigInt
11
12   def toBigInt: BigInt = underlying
13
14   /** Factory function to create the underlying T, for instance a
         UInt32 */
15   def apply: BigInt => Int64
16
17   def +(num: Int64): Int64 = apply(checkResult(underlying + num.
         underlying))
18
19   /**
20     * Checks if the given result is within the range
21     * of this number type
22     */
23   private def checkResult(result: BigInt): BigInt = {
24     require(
25          result <= BigInt("9223372036854775807")
26       && result >= BigInt("-9223372036854775808"))
27     result
28   }
29 }
30
31 /**
32   * Represents a signed number in our number system
33   * Instances of this is [[Int64]]
34   */
35 sealed abstract class SignedNumber extends Number
36
```

```
/**
  * Represents a int64_t in C
  */
sealed abstract class Int64 extends SignedNumber {
  override def apply: BigInt => Int64 = Int64(_)
}

/**
  * Represents various numbers that should be implemented
  * inside of any companion object for a number
  */
trait BaseNumbers[T] {
  def zero: T
  def one: T
  def min: T
  def max: T
}

case object Int64 extends BaseNumbers[Int64] {
  lazy val zero = Int64(0)
  lazy val one = Int64(1)

  lazy val min = Int64(BigInt("-9223372036854775808"))
  lazy val max = Int64(BigInt("9223372036854775807"))

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}

private case class Int64Impl(underlying: BigInt) extends Int64 {
  require(underlying >= BigInt("-9223372036854775808"))
  require(underlying <= BigInt("9223372036854775807"))
}
```

```
package addition.modified.currency

import addition.modified.number.{BaseNumbers, Int64}
import stainless.lang._

sealed abstract class CurrencyUnit {
  def satoshis: Satoshis

  def !=(c: CurrencyUnit): Boolean = !(this == c)

  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis

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

```
18
19    protected def underlying: Int64}
20
21  sealed abstract class Satoshis extends CurrencyUnit {
22    override def satoshis: Satoshis = this
23
24    def toBigInt: BigInt = underlying.toBigInt
25
26    def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
          underlying
27  }
28
29  case object Satoshis extends BaseNumbers[Satoshis] {
30    val min = Satoshis(Int64.min)
31    val max = Satoshis(Int64.max)
32    val zero = Satoshis(Int64.zero)
33    val one = Satoshis(Int64.one)
34
35    def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
36  }
37
38  private case class SatoshisImpl(underlying: Int64) extends Satoshis
```

4 Conclusion

Because of the limitations of the verication tool, we could only verify a rewritten version of the original Bitcoin-S code. So we can not guarantee the correctness of the addition of Satoshis with zero in Bitcoin-S. Not all changes we made were as trivial as the replacement of objects with case objects. For these non-trivial changes, as seen for example the bound check in section B.7, we cannot say whether they are equivalent to the original implementation or not.

So code should be written specically with formal verication in mind, in order to successfully verify it. Otherwise, it needs a lot of changes in the software because verification is mathematical and the current software is written mostly in object-oriented style. Software written in the functional paradigm would be much easier to reason about.

Thus, either Stainless must find ways to translate more of built-in object-oriented patterns of Scala to their verification tool or developers must invest more in functional programming.

Also, we found that trying to verify code reveals bugs as shown in section **??**. Finally, our work led to some feedback to the Stainless developers to improve the tool.

conclusions: what's future work? how to change bitcoin-s? how to extend stainless?

References

1. Blanc, R., Kuncak, V.: Sound reasoning about integral data types with a reusable SMT solver interface. In: Haller and Miller [4], 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. developers, T.S.: The stainless repository, https://github.com/epfl-lara/stainless, accessed 2019-06-19
4. 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)
5. 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
6. Stainless documentation: https://epfl-lara.github.io/stainless/, accessed 2019-06-19
7. Suredbits & the bitcoin-s developers: The bitcoin-s website, https://bitcoin-s.org, accessed 2019-06-19
8. The bitcoin-s developers: The bitcoin-s repository, https://github.com/bitcoin-s, accessed 2019-06-19
9. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Haller and Miller [4], pp. 18–29. https://doi.org/10.1145/2774975.2774978

A Project Setup

In this chapter we look at the practical challenges that could occur by using Stainless. Since Stainless is still in development and there is no 1.x release there might still be breaking changes and improvements fixing problems described now.

nur über stainless-sbt-plugin reden damit man weiss wir haben es versucht

A.1 stainless-sbt-plugin vs JAR

We can either use the sbt plugin or a JAR file to check code with Stainless.

Invoking the JAR on our source code Stainless will verify it. If we have a bigger project, this becomes really tricky, because we must pass all files needed including the dependencies. This is in contrast to the sbt plugin, where we can integrate Stainless in our compilation process. When we call compile, Stainless verifies the code and stops the compilation if the verification fails.

Having a static version configured in the sbt build file, every developer has the same Stainless features available. This should prevent incompatibility with new or deprecated features when we use different plugins.

So the sbt plugin has clear advantages over the JAR file since its integrated directly. We do not have to download it manually and find the right version and

if we bump the version we can just edit it in the build file and every developer is on the same version again.

However, currently there are some drawbacks. For example the sbt plugin does not always report errors.

We use the jar for everything.

A.2 Integration into Bitcoin-S

During this work, Stainless updated the sbt plugin to support sbt 1.2.8 from 0.13.17 and Scala 2.12.8 from 2.11.12. So this section might be out of date now.

Stainless requires and Scala recommends Java SE Development Kit 8. Newer Java versions won't work.

To use the latest version of the sbt tool you have to build it locally. You can run `sbt universal:stage` in the cloned Stainless git repository. This generates *frontends/scalac/target/universal/stage/bin/stainless-scalac*.

Bitcoin-S-Core uses sbt 1.2.8 and Scala 2.12.8, while Stainless sbt plugin is on sbt 0.13.17 and Scala 2.11.12.

Sbt introduced new features in the 1.x release used by Bitcoin-S. Most of them can be written the sbt 0.13.17 way.

The bigger problem is, due to the different Scala and sbt versions, the following error after trying to go in a sbt shell:

```
[warn] There may be incompatibilities among your library dependencies; run 'evicted'
       to see detailed eviction warnings.
[error] java.lang.NoClassDefFoundError: sbt/SourcePosition
...
Project loading failed: (r)etry, (q)uit, (l)ast, or (i)gnore?
```

Downgrading Bitcoin-S sbt version to 0.13.17 fixes the error but then it can not load some libraries only compiled for newer versions. So this would take too much time to fix and changes the Bitcoin-S code inadvertently.

The next approach is to use the stainless cli instead of sbt. Running stainless on all source files does not work, because dependencies are missing. The parameter *-classpath* can resolve it but the value of this parameter must be the paths to all the dependencies separated by a ':'. Finally, *core* depends on *secp256k1jni*, another package of Bitcoin-S written in Java. So this needs to be in the source files to.

The final command looks like this in *core* folder of Bitcoin-S:

```
$ stainless
  -classpath ".:$(find ~/.ivy2/ -type f -name *.jar | tr '\n' ':')"
  $(find . -type f -name *.scala | tr '\n' ' ')
  $(find ../secp256k1jni -type f -name *.java | tr '\n' ' ')
```

.ivy2 is the dependency cache of sbt. The *tr* replaces the first char with the second so a newline with either ':' or ' '.

With this command, Stainless throws the next error:

```
[Internal] Error: object scala.reflect.macros.internal.macroImpl in compiler mirror
           not found.. Trace:
[Internal] - scala.reflect.internal.MissingRequirementError$.signal
           (MissingRequirementError.scala:17)
...
[Internal] object scala.reflect.macros.internal.macroImpl in compiler mirror not found.
[Internal] Please inform the authors of Inox about this message
```

So we can not know how many errors will face us. Let's go another way, because the errors may take too much time and it might lead to a next error. We extract the code needed to verify a transaction mainly the class Transaction and ScriptInterpreter with many other classes they're depending on.

After this extraction Stainless was successfully integrated with both sbt and JAR.

Running `sbt compile` in the project with Stainless ended without error. But it also ended with no output. So we are not able to change the code so Stainless would accept it since we do not know what to change.

So the sbt plugin does not always complain where the JAR file did. The open issue 484 on GitHub might describe exactly this error.

Now we can finally run Stainless on our code. But this leads us to the next findings. We must rewrite most of the code, as described in the previous chapters.

- clone our repo
- jar vs sbt, you can use both
- contains a stainless jar
- call the script blahblah

B Code Transformations

These are the code transformations needed to Verify the Addition-with-Zero Property.

When we run Stainless on this code (without any properties to prove), it throws the following errors:

```
[ Error  ] currency/CurrencyUnits.scala:6:3: Stainless doesn't
             support abstract type members
           type A
           ^^^^^^
[ Error  ] currency/CurrencyUnits.scala:26:33: Only literal arguments
             are allowed for BigInt.
           def toBigInt: BigInt = BigInt(toLong)
                                         ^^^^^^
[ Error  ] currency/CurrencyUnits.scala:33:1: Objects cannot extend
             classes or implement traits, use a case object instead
         object Satoshis extends BaseNumbers[Satoshis] {
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] number/NumberType.scala:65:1: Objects cannot extend
             classes or implement traits, use a case object instead
```

```
          object Int64 extends BaseNumbers[Int64] {
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[  Info  ] Shutting down executor service.
```

So let's see how we can fix those errors.

B.1 An Inheriting Object

Stainless output:

```
[ Error  ] currency/CurrencyUnits.scala:33:1: Objects cannot extend
             classes or implement traits, use a case object instead
          object Satoshis extends BaseNumbers[Satoshis] {
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] number/NumberType.scala:65:1: Objects cannot extend
             classes or implement traits, use a case object instead
          object Int64 extends BaseNumbers[Int64] {
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Code before:

```scala
object Int64 extends BaseNumbers[Int64] { ... }
object Satoshis extends BaseNumbers[Satoshis] { ... }
```

Code after:

```scala
case object Int64 extends BaseNumbers[Int64] { ... }
case object Satoshis extends BaseNumbers[Satoshis] { ... }
```

Here, we can just change the objects from object to case object. Stainless recommendation is to use objects for modules and case objects as algebraic data types.

Kai this does not change the semantic

This is due to the internal design of Scala. It's possible to reason about case object but not about object. This needs a fundamental knowledge of Scala and some functional paradigms that should not be part of this thesis. The issue number 520 on Stainless GitHub gives some thoughts, if you want to know more.

B.2 Abstract Type Member

Stainless output:

```
[ Error  ] currency/CurrencyUnits.scala:6:3: Stainless doesn't
             support abstract type members
            type A
            ^^^^^^
```

This should be easy to rewrite by using generics instead of an abstract type, right? Unfortunately not. The problem is, CurrencyUnit uses one of its implementing classes: Satoshis.

kai this does not change the semantic

Before:

```
sealed abstract class CurrencyUnit {
  type A

  def satoshis: Satoshis

  def !=(c: CurrencyUnit): Boolean = !(this == c)

  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
}
```

What happens, if we typify CurrencyUnit with A, meaning to make it generic with type A?

Satoshis extends CurrencyUnit with type Int64, so it would be of type CurrencyUnit[Int64]. That's too specific, because the return type of the addition is then CurrencyUnit[Int64] not CurrencyUnit[A]. Maybe the Bitcoin-S developers should reimplement this part and not use Satoshis directly.

Since there is no easy way to fix it and the code should stay as much as possible the original, we just remove the abstract type and set it to Int64. This limits the verification a bit, but as we only want to verify the addition in satoshis, that's OK.

After:

```
sealed abstract class CurrencyUnit {
  def satoshis: Satoshis

  def !=(c: CurrencyUnit): Boolean = !(this == c)

  def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis

  def +(c: CurrencyUnit): CurrencyUnit = {
    Satoshis(satoshis.underlying + c.satoshis.underlying)
```

```
10    }
11
12    protected def underlying: Int64
13  }
14
15  sealed abstract class Satoshis extends CurrencyUnit {
16    override def satoshis: Satoshis = this
17
18    def toBigInt: BigInt = BigInt(toLong)
19
20    def toLong: Long = underlying.toLong
21
22    def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
          underlying
23  }
```

B.3 Non-Literal BigInt Constructor Argument

Stainless output:

```
[ Error  ] currency/CurrencyUnits.scala:26:33: Only literal arguments
             are allowed for BigInt.
           def toBigInt: BigInt = BigInt(toLong)
                                         ^^^^^^
```

As described before, Stainless supports only a subset of Scala. The BigInt from the Stainless library is a bit restricted. One such restriction is, that BigInt does not support dynamic BigInt construction. Thus, the constructor parameter of BitInt must be a literal argument.

kai this does not change semantic

Before:

```
1  sealed abstract class Satoshis extends CurrencyUnit {
2    override def satoshis: Satoshis = this
3
4    def toBigInt: BigInt = BigInt(toLong)
5
6    def toLong: Long = underlying.toLong
7
8    def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
         underlying
9  }
```

After:

```
1  sealed abstract class Satoshis extends CurrencyUnit {
2    override def satoshis: Satoshis = this
3
4    def toBigInt: BigInt = underlying.toBigInt
5
6    def toLong: Long = underlying.toLong
```

```
7
8   def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
        underlying
9 }
```

This would be really hard to refactor, because Bitcoin-S tries to be as dynamic as possible so it can be used with cryptocurrencies other than bitcoins. Maybe it could be impossible, because they need to parse dynamic values from the bitcoin network.

Luckily, we can use toBigInt on the field `underlying` directly instead of toLong. So, instead of converting the underlying to Long and back to BigInt we convert underlying directly to BigInt.

After fixing all Stainless errors, a new error appears.

B.4 Self-Reference in Type Parameter Bound

Stainless output:

```
[ Error   ] number/NumberType.scala:8:30: Unknown type parameter type
              T
            sealed abstract class Number[T <: Number[T]] {
                                           ^^^^^^^^^^^^^^
```

kai this does not change semantic

This is a missing feature in Stainless. It does not support upper type boundaries on the class itself. To track this, issue 519 was created on GitHub during this work.

Before:

```
1  sealed abstract class Number[T <: Number[T]] {
2    type A = BigInt
3
4    /** The underlying scala number used to to hold the number */
5    protected def underlying: A
6
7    def toLong: Long = toBigInt.bigInteger.longValueExact()
8    def toBigInt: BigInt = underlying
9
10   /**
11     * This is used to determine the valid amount of bytes in a number
12     * for instance a UInt8 has an andMask of 0xff
13     * a UInt32 has an andMask of 0xffffffff
14     */
15   def andMask: BigInt
16
17   /** Factory function to create the underlying T, for instance a
         UInt32 */
18   def apply: A => T
19
20   def +(num: T): T = apply(checkResult(underlying + num.underlying))
```

```
21
22    /**
23      * Checks if the given result is within the range
24      * of this number type
25      */
26    private def checkResult(result: BigInt): A = {
27      require((result & andMask) == result,
28        "Result was out of bounds, got: " + result)
29      result
30    }
31  }
32
33  /**
34    * Represents a signed number in our number system
35    * Instances of this is [[Int64]]
36    */
37  sealed abstract class SignedNumber[T <: Number[T]] extends Number[T]
38
39  /**
40    * Represents a int64_t in C
41    */
42  sealed abstract class Int64 extends SignedNumber[Int64] {
43    override def apply: A => Int64 = Int64(_)
44    override def andMask = 0xffffffffffffffffL
45  }
```

After:

```
1   sealed abstract class Number {
2     type A = BigInt
3
4     /** The underlying scala number used to to hold the number */
5     protected def underlying: A
6
7     def toLong: Long = toBigInt.bigInteger.longValueExact()
8     def toBigInt: BigInt = underlying
9
10    /**
11      * This is used to determine the valid amount of bytes in a number
12      * for instance a UInt8 has an andMask of 0xff
13      * a UInt32 has an andMask of 0xffffffff
14      */
15    def andMask: BigInt
16
17    /** Factory function to create the underlying T, for instance a
          UInt32 */
18    def apply: A => Int64
19
20    def +(num: Int64): Int64 = apply(checkResult(underlying + num.
          underlying))
21
```

```
22    /**
23      * Checks if the given result is within the range
24      * of this number type
25      */
26    private def checkResult(result: BigInt): A = {
27      require((result & andMask) == result,
28        "Result was out of bounds, got: " + result)
29      result
30    }
31  }
32
33  /**
34    * Represents a signed number in our number system
35    * Instances of this is [[Int64]]
36    */
37  sealed abstract class SignedNumber extends Number
38
39  /**
40    * Represents a int64_t in C
41    */
42  sealed abstract class Int64 extends SignedNumber {
43    override def apply: A => Int64 = Int64(_)
44    override def andMask = 0xffffffffffffffffL
45  }
```

Despite this, in order to be able to continue, we make this a concrete type by replacing T with Int64. Int64, because Satoshis uses only Int64. There are other number types like UInt16 but for our property we don't need them.

Now, there are two new errors.

B.5 Missing Member `bigInteger` in BigInt

Ramon is .bigInteger missing in Stainless? Stainless output:

```
[ Error  ] number/NumberType.scala:14:22: Unknown call to bigInteger
             on Number.this.toBigInt (BigInt) with arguments List()
             of type List()
           def toLong: Long = toBigInt.bigInteger.longValueExact()
                              ^^^^^^^^^^^^^^^^^^^
```

kai This does not change semantic

Now that we do not depend on toLong anymore lets just remove it from out code.

Before:

```
1  sealed abstract class Satoshis extends CurrencyUnit {
2    override def satoshis: Satoshis = this
3
4    def toBigInt: BigInt = underlying.toBigInt
5
```

```
  def toLong: Long = underlying.toLong

  def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
      underlying
}
```

After:

```
sealed abstract class Satoshis extends CurrencyUnit {
  override def satoshis: Satoshis = this

  def toBigInt: BigInt = underlying.toBigInt

  def ==(satoshis: Satoshis): Boolean = underlying == satoshis.
      underlying
}
```

And before:

```
sealed abstract class Number {
  type A = BigInt

  /** The underlying scala number used to to hold the number */
  protected def underlying: A

  def toLong: Long = toBigInt.bigInteger.longValueExact()
  def toBigInt: BigInt = underlying

  /**
    * This is used to determine the valid amount of bytes in a number
    * for instance a UInt8 has an andMask of 0xff
    * a UInt32 has an andMask of 0xffffffff
    */
  def andMask: BigInt

  /** Factory function to create the underlying T, for instance a
      UInt32 */
  def apply: A => Int64

  def +(num: Int64): Int64 = apply(checkResult(underlying + num.
      underlying))

  /**
    * Checks if the given result is within the range
    * of this number type
    */
  private def checkResult(result: BigInt): A = {
    require((result & andMask) == result,
      "Result␣was␣out␣of␣bounds,␣got:␣" + result)
    result
  }
}
```

After:

```
sealed abstract class Number {
  type A = BigInt

  /** The underlying scala number used to to hold the number */
  protected def underlying: A

  def toBigInt: BigInt = underlying

  /**
    * This is used to determine the valid amount of bytes in a number
    * for instance a UInt8 has an andMask of 0xff
    * a UInt32 has an andMask of 0xffffffff
    */
  def andMask: BigInt

  /** Factory function to create the underlying T, for instance a
      UInt32 */
  def apply: A => Int64

  def +(num: Int64): Int64 = apply(checkResult(underlying + num.
      underlying))

  /**
    * Checks if the given result is within the range
    * of this number type
    */
  private def checkResult(result: BigInt): A = {
    require((result & andMask) == result,
      "Result was out of bounds, got: " + result)
    result
  }
}
```

B.6 Type Member

Stainless output:

```
[Warning ] number/NumberType.scala:9:3: Could not extract tree in
             class: type A = BigInt (class scala.reflect.internal.
             Trees$TypeDef)
               type A = BigInt
               ^^^^^^^^^^^^^^^
```

kai this does not change semantic

Before:

```
sealed abstract class Number {
  type A = BigInt

```

```
  /** The underlying scala number used to to hold the number */
  protected def underlying: A

  def toBigInt: BigInt = underlying

  /** Factory function to create the underlying T, for instance a
      UInt32 */
  def apply: A => Int64

  def +(num: Int64): Int64 = apply(checkResult(underlying + num.
      underlying))

  /**
    * Checks if the given result is within the range
    * of this number type
    */
  private def checkResult(result: BigInt): A = {
    require(
          result <= BigInt("9223372036854775807")
      && result >= BigInt("-9223372036854775808"))
    result
  }
}

/**
  * Represents a signed number in our number system
  * Instances of this is [[Int64]]
  */
sealed abstract class SignedNumber extends Number

/**
  * Represents a int64_t in C
  */
sealed abstract class Int64 extends SignedNumber {
  override def apply: A => Int64 = Int64(_)
}
```

After:

```
sealed abstract class Number {
  /** The underlying scala number used to to hold the number */
  protected def underlying: BigInt

  def toBigInt: BigInt = underlying

  /** Factory function to create the underlying T, for instance a
      UInt32 */
  def apply: BigInt => Int64

  def +(num: Int64): Int64 = apply(checkResult(underlying + num.
      underlying))
```

```
11
12    /**
13      * Checks if the given result is within the range
14      * of this number type
15      */
16    private def checkResult(result: BigInt): BigInt = {
17      require(
18           result <= BigInt("9223372036854775807")
19        && result >= BigInt("-9223372036854775808"))
20      result
21    }
22  }
23
24  /**
25    * Represents a signed number in our number system
26    * Instances of this is [[Int64]]
27    */
28  sealed abstract class SignedNumber extends Number
29
30  /**
31    * Represents a int64_t in C
32    */
33  sealed abstract class Int64 extends SignedNumber {
34    override def apply: BigInt => Int64 = Int64(_)
35  }
```

This is easy. We just replace all occurrence of A with BigInt, since A is not overwritten in an implementing class. This is not the exact same code, because an implementing class can not override A anymore but that's fine for our verification.

This was a missing feature in Stainless that was fixed on Mai 28, 2019 with pull request 470 on GitHub. Now it should work without this change.

B.7 Missing Bitwise-And Method on BigInt

Stainless output:

```
[ Error   ] number/NumberType.scala:33:14: Unknown call to & on result
              (BigInt) with arguments List(Number.this.andMask) of
             type List(BigInt)
               require((result & andMask) == result,
                       ^^^^^^^^^^^^^^^^
```

kai here we do not know if it changes semantic

Due to the restrictions on BigInt, we can not use the & function either. Before:

```
1  sealed abstract class Number {
2    type A = BigInt
3
```

```
4   /** The underlying scala number used to to hold the number */
5   protected def underlying: A
6
7   def toBigInt: BigInt = underlying
8
9   /**
10    * This is used to determine the valid amount of bytes in a number
11    * for instance a UInt8 has an andMask of 0xff
12    * a UInt32 has an andMask of 0xffffffff
13    */
14  def andMask: BigInt
15
16  /** Factory function to create the underlying T, for instance a
        UInt32 */
17  def apply: A => Int64
18
19  def +(num: Int64): Int64 = apply(checkResult(underlying + num.
        underlying))
20
21  /**
22    * Checks if the given result is within the range
23    * of this number type
24    */
25  private def checkResult(result: BigInt): A = {
26    require((result & andMask) == result,
27      "Result was out of bounds, got: " + result)
28    result
29  }
30 }
31
32 /**
33   * Represents a signed number in our number system
34   * Instances of this is [[Int64]]
35   */
36 sealed abstract class SignedNumber extends Number
37
38 /**
39   * Represents a int64_t in C
40   */
41 sealed abstract class Int64 extends SignedNumber {
42   override def apply: A => Int64 = Int64(_)
43   override def andMask = 0xffffffffffffffffL
44 }
```

After:

```
1  sealed abstract class Number {
2    type A = BigInt
3
4    /** The underlying scala number used to to hold the number */
5    protected def underlying: A
```

```
6
7     def toBigInt: BigInt = underlying
8
9     /** Factory function to create the underlying T, for instance a
          UInt32 */
10    def apply: A => Int64
11
12    def +(num: Int64): Int64 = apply(checkResult(underlying + num.
          underlying))
13
14    /**
15      * Checks if the given result is within the range
16      * of this number type
17      */
18    private def checkResult(result: BigInt): A = {
19      require(
20            result <= BigInt("9223372036854775807")
21        && result >= BigInt("-9223372036854775808"),
22        "Result was out of bounds, got: " + result)
23      result
24    }
25  }
26
27  /**
28    * Represents a signed number in our number system
29    * Instances of this is [[Int64]]
30    */
31  sealed abstract class SignedNumber extends Number
32
33  /**
34    * Represents a int64_t in C
35    */
36  sealed abstract class Int64 extends SignedNumber {
37    override def apply: A => Int64 = Int64(_)
38  }
```

This is a bounds check. It checks if the result of the addition is in range of the specified type, which is now the hard coded Int64.

So, we can replace the & mask with a bound check whether the result is in range of Long.MinValue and Long.MaxValue, because Int64 has the same 64-bit range as Long. Again the code gets a bit more static and it's not the exact same code anymore.

Running Stainless produces again new errors.

B.8 Inner Class in Case Object

Stainless output:

```
[Warning ] currency/CurrencyUnits.scala:36:3: Could not extract tree
             in class: case private class SatoshisImpl extends
             Satoshis with Product with Serializable {
...
[Warning ] } (class scala.reflect.internal.Trees$ClassDef)
              private case class SatoshisImpl(underlying: Int64)
                          extends Satoshis
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

And:

```
[Warning ] number/NumberType.scala:59:3: Could not extract tree in
             class: case private class Int64Impl extends Int64 with
             Product with Serializable {
...
[Warning ] } (class scala.reflect.internal.Trees$ClassDef)
              private case class Int64Impl(underlying: BigInt)
                          extends Int64 {
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
```

Stainless can not extract the private class inside the object. Bitcoin-S uses this a lot, because they separate the class from its implementation.

kai this does not change semantic

Before:

```scala
case object Satoshis extends BaseNumbers[Satoshis] {
  val min = Satoshis(Int64.min)
  val max = Satoshis(Int64.max)
  val zero = Satoshis(Int64.zero)
  val one = Satoshis(Int64.one)

  def apply(int64: Int64): Satoshis = SatoshisImpl(int64)

  private case class SatoshisImpl(underlying: Int64) extends Satoshis
}
```

After:

```scala
case object Satoshis extends BaseNumbers[Satoshis] {

  val min = Satoshis(Int64.min)
  val max = Satoshis(Int64.max)
  val zero = Satoshis(Int64.zero)
  val one = Satoshis(Int64.one)

  def apply(int64: Int64): Satoshis = SatoshisImpl(int64)
}

private case class SatoshisImpl(underlying: Int64) extends Satoshis
```

And before:

```
case 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)
  lazy val one = Int64(1)

  lazy val min = Int64(-9223372036854775808L)
  lazy val max = Int64(9223372036854775807L)

  def apply(long: Long): Int64 = Int64(BigInt(long))

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

After:

```
case object Int64 extends BaseNumbers[Int64] {
  lazy val zero = Int64(0)
  lazy val one = Int64(1)

  lazy val min = Int64(-9223372036854775808L)
  lazy val max = Int64(9223372036854775807L)

  def apply(long: Long): Int64 = Int64(BigInt(long))

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}

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)
}
```

This is easy to fix. We just extract the private class out of the object. This is not exactly the same code, because other classes in the same file could now access the private class. But for our property it does not change anything.

Now we get some weird warnings about require.

B.9 Message Parameter in Require

Stainless output:

```
[Warning ] number/NumberType.scala:9:3: Could not extract tree in
             class: type A = BigInt (class scala.reflect.internal.
             Trees$TypeDef)
```

```
          type A = BigInt
          ^^^^^^^^^^^^^^^
[Warning ] number/NumberType.scala:71:3: Could not extract tree in
             class: scala.this.Predef.require(Int64Impl.this.
             underlying.>=(math.this.BigInt.long2bigInt
             (-92233720368547758O8L)), "Number was too small for a
             int64, got: ".+(Int64Impl.this.underlying)) (class scala
             .reflect.internal.Trees$Apply)
          require(underlying >= -9223372036854775808L,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[Warning ] number/NumberType.scala:73:3: Could not extract tree in
             class: scala.this.Predef.require(Int64Impl.this.
             underlying.<=(math.this.BigInt.long2bigInt
             (9223372036854775807L)), "Number was too big for a int64
             , got: ".+(Int64Impl.this.underlying)) (class scala.
             reflect.internal.Trees$Apply)
          require(underlying <= 9223372036854775807L,
          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^...
[ Error  ] checkResult$0 depends on missing dependencies: require$1.
```

Seems like Stainless does not support the second string parameter of require or at least it throws a warning about it. We can safely remove the string parameters from the requires, since they only serve as error messages.

kai this does not change semantic

Before:

```
1 private case class Int64Impl(underlying: BigInt) extends Int64 {
2   require(underlying >= -9223372036854775808L,
3     "Number␣was␣too␣small␣for␣a␣int64,␣got:␣" + underlying)
4   require(underlying <= 9223372036854775807L,
5     "Number␣was␣too␣big␣for␣a␣int64,␣got:␣" + underlying)
6 }
```

After:

```
1 private case class Int64Impl(underlying: BigInt) extends Int64 {
2   require(underlying >= -9223372036854775808L)
3   require(underlying <= 9223372036854775807L)
4 }
```

And before:

```
1 sealed abstract class Number {
2   type A = BigInt
3
4   /** The underlying scala number used to to hold the number */
5   protected def underlying: A
6
7   def toBigInt: BigInt = underlying
8
9   /** Factory function to create the underlying T, for instance a
        UInt32 */
```

```
10    def apply: A => Int64
11
12    def +(num: Int64): Int64 = apply(checkResult(underlying + num.
          underlying))
13
14    /**
15      * Checks if the given result is within the range
16      * of this number type
17      */
18    private def checkResult(result: BigInt): A = {
19      require(
20           result <= BigInt("9223372036854775807")
21        && result >= BigInt("-9223372036854775808"),
22        "Result was out of bounds, got: " + result)
23      result
24    }
25  }
```

After:

```
1   sealed abstract class Number {
2     type A = BigInt
3
4     /** The underlying scala number used to to hold the number */
5     protected def underlying: A
6
7     def toBigInt: BigInt = underlying
8
9     /** Factory function to create the underlying T, for instance a
          UInt32 */
10    def apply: A => Int64
11
12    def +(num: Int64): Int64 = apply(checkResult(underlying + num.
          underlying))
13
14    /**
15      * Checks if the given result is within the range
16      * of this number type
17      */
18    private def checkResult(result: BigInt): A = {
19      require(
20           result <= BigInt("9223372036854775807")
21        && result >= BigInt("-9223372036854775808"))
22      result
23    }
24  }
```

A new error appears.

B.10 Missing Implicit Long to BigInt Conversion

Stainless output:

```
[ Error  ] inv$4 depends on missing dependencies: long2bigInt$0.
[ Error  ] apply$14 depends on missing dependencies: BigInt$0,
             apply$15.
```

This error is hard to understand, but we can see that there is a missing Long to BigInt conversion. So we search for all Long values in the code.

kai this does not change semantic

Before:

```
private case class Int64Impl(underlying: BigInt) extends Int64 {
  require(underlying >= -9223372036854775808L)
  require(underlying <= 9223372036854775807L)
}
```

After:

```
private case class Int64Impl(underlying: BigInt) extends Int64 {
  require(underlying >= BigInt("-9223372036854775808"))
  require(underlying <= BigInt("9223372036854775807"))
}
```

And before:

```
case object Int64 extends BaseNumbers[Int64] {
  lazy val zero = Int64(0)
  lazy val one = Int64(1)

  lazy val min = Int64(-9223372036854775808L)
  lazy val max = Int64(9223372036854775807L)

  def apply(long: Long): Int64 = Int64(BigInt(long))

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

After:

```
case object Int64 extends BaseNumbers[Int64] {
  lazy val zero = Int64(0)
  lazy val one = Int64(1)

  lazy val min = Int64(BigInt("-9223372036854775808"))
  lazy val max = Int64(BigInt("9223372036854775807"))

  def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
}
```

Looks like it can not compare a BigInt with a Long value. We can easily convert this Long value to a BigInt with a string literal parameter by passing these numbers as strings to the BigInt constructor.

Finally, we get some output from Stainless about the verification in the code.

```
[  Info  ]  - Checking cache: 'adt invariant' VC for apply @33:39...
[  Info  ]  - Checking cache: 'adt invariant' VC for apply @62:38...
[  Info  ] Cache miss: 'adt invariant' VC for apply @62:38...
[  Info  ] Cache hit: 'adt invariant' VC for apply @33:39...
[  Info  ]  - Now solving 'adt invariant' VC for apply @62:38...
[  Info  ]  - Result for 'adt invariant' VC for apply @62:38:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   bigInt: BigInt -> -9223372036854775809
[  Info  ]  - Checking cache: 'precond. (call checkResult(thiss,
             underlying(thiss) + u ...)' VC for + @17:36...
[  Info  ] Cache miss: 'precond. (call checkResult(thiss, underlying(
             thiss) + u ...)' VC for + @17:36...
[  Info  ]  - Now solving 'precond. (call checkResult(thiss,
             underlying(thiss) + u ...)' VC for + @17:36...
[  Info  ]  - Result for 'precond. (call checkResult(thiss,
             underlying(thiss) + u ...)' VC for + @17:36:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   num: Number   -> Int64Impl(9223372036854775807)
[Warning ]   thiss: Number -> Int64Impl(1)
```

This shows that there is an invalid specification in checkResult and Stainless prints a counterexample for it.

Let's ignore this for a moment and write the specification for the **??**.

B.11 Missing BigInt Constructor with Long Argument

Split code from above

B.12 Writing Specification for the Property

As specified, our verification must only support addition with zero. So we restrict the parameter to be zero in the precondition.

```
1  require(c.satoshis == Satoshis.zero)
```

We ensure the result is the same value as *this* in the postcondition.

```
1  ensuring(res => res.satoshis == this.satoshis)
```

We can use equals (==) directly on Satoshis, because it is a case class. Before:

```
1  sealed abstract class CurrencyUnit {
2    def satoshis: Satoshis
3
4    def !=(c: CurrencyUnit): Boolean = !(this == c)
5
6    def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
7
```

```
8   def +(c: CurrencyUnit): CurrencyUnit = {
9     Satoshis(satoshis.underlying + c.satoshis.underlying)
10  }
11
12  protected def underlying: Int64
13 }
```

The addition will now look like this:

```
1  sealed abstract class CurrencyUnit {
2    def satoshis: Satoshis
3
4    def !=(c: CurrencyUnit): Boolean = !(this == c)
5
6    def ==(c: CurrencyUnit): Boolean = satoshis == c.satoshis
7
8    override def +(c: CurrencyUnit): CurrencyUnit = {
9      require(c.satoshis == Satoshis.zero)
10     Satoshis(satoshis.underlying + c.satoshis.underlying)
11   } ensuring(res => res.satoshis == this.satoshis)
12
13   protected def underlying: Int64
14 }
```

That's all we need to verify our addition.
Now we will look into the previous error.

B.13 Propagating Require

There is another problem with Bitcoin-S. Bitcoin-S-Core uses require as a fail-fast method whereas Stainless needs it to verify the code.

Stainless output now:

```
[  Info  ]  - Checking cache: 'postcondition' VC for + @12:3...
[  Info  ]  - Checking cache: 'adt invariant' VC for apply @34:39...
[  Info  ]  - Checking cache: 'adt invariant' VC for apply @62:38...
[  Info  ] Cache miss: 'adt invariant' VC for apply @62:38...
[  Info  ] Cache hit: 'adt invariant' VC for apply @34:39...
[  Info  ] Cache hit: 'postcondition' VC for + @12:3...
[  Info  ]  - Now solving 'adt invariant' VC for apply @62:38...
[  Info  ]  - Result for 'adt invariant' VC for apply @62:38:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   bigInt: BigInt -> -9223372036854775809
[  Info  ]  - Checking cache: 'precond. (call checkResult(thiss,
             underlying(thiss) + u ...)' VC for + @17:36...
[  Info  ] Cache miss: 'precond. (call checkResult(thiss, underlying(
             thiss) + u ...)' VC for + @17:36...
[  Info  ]  - Now solving 'precond. (call checkResult(thiss,
             underlying(thiss) + u ...)' VC for + @17:36...
```

```
[  Info  ]  - Result for 'precond. (call checkResult(thiss,
              underlying(thiss) + u ...)' VC for + @17:36:
[Warning ]  => INVALID
[Warning ] Found counter-example:
[Warning ]   num: Number   -> Int64Impl(9223372036854775807)
[Warning ]   thiss: Number -> Int64Impl(1)
```

Corresponding code:

```
sealed abstract class Number {
  /** The underlying scala number used to to hold the number */
  protected def underlying: BigInt

  def toBigInt: BigInt = underlying

  /** Factory function to create the underlying T, for instance a
      UInt32 */
  def apply: BigInt => Int64

  def +(num: Int64): Int64 = apply(checkResult(underlying + num.
      underlying))

  /**
    * Checks if the given result is within the range
    * of this number type
    */
  private def checkResult(result: BigInt): BigInt = {
    require(
         result <= BigInt("9223372036854775807")
      && result >= BigInt("-9223372036854775808"))
    result
  }
}
```

But how does Stainless find a counter example ignoring the require in checkResult? Since Stainless is a static verification tool, it tests every possibility. So it can use a number bigger than the maximum Int64 and pass it to the addition. The require in checkResult fails.

Thus, we need to add the restriction from checkResult to the addition too.

```
override def +(num: Int64): Int64 = {
  require(
       num.underlying <= BigInt("9223372036854775807")
    && num.underlying >= BigInt("-9223372036854775808")
    && this.underlying <= BigInt("9223372036854775807")
    && this.underlying >= BigInt("-9223372036854775808")
  )
  apply(checkResult(underlying + num.underlying))
}
```

Stainless finds another counter example:

```
[Warning ] Found counter-example:
[Warning ]   num: { x: Object | @unchecked isInt64(x) }    ->
             Int64Impl(1)
[Warning ]   thiss: { x: Object | @unchecked isNumber(x) } ->
                Int64Impl(9223372036854775807)
```

Sure, when adding one to the maximum Int64 the require does not hold anymore. Since we do only allow zero as a parameter, the easiest way is to restrict it to zero here too.

Code after:

```
sealed abstract class Number {
  /** The underlying scala number used to to hold the number */
  protected def underlying: BigInt

  def toBigInt: BigInt = underlying

  /** Factory function to create the underlying T, for instance a
      UInt32 */
  def apply: BigInt => Int64

  def +(num: Int64): Int64 = {
    require(
          num == Int64.zero
      && this.underlying <= BigInt("9223372036854775807")
      && this.underlying >= BigInt("-9223372036854775808"))
    apply(checkResult(underlying + num.underlying))
  }

  /**
    * Checks if the given result is within the range
    * of this number type
    */
  private def checkResult(result: BigInt): BigInt = {
    require(
          result <= BigInt("9223372036854775807")
      && result >= BigInt("-9223372036854775808"))
    result
  }
}
```

The same problem occurs in the Int64 constructor.

Before:

```
sealed abstract class Int64 extends SignedNumber {
  override def apply: BigInt => Int64 = Int64(_)
}

case object Int64 extends BaseNumbers[Int64] {
  lazy val zero = Int64(0)
  lazy val one = Int64(1)
```

```
8
9     lazy val min = Int64(BigInt("-9223372036854775808"))
10    lazy val max = Int64(BigInt("9223372036854775807"))
11
12    def apply(bigInt: BigInt): Int64 = Int64Impl(bigInt)
13  }
14
15  private case class Int64Impl(underlying: BigInt) extends Int64 {
16    require(underlying >= BigInt("-9223372036854775808"))
17    require(underlying <= BigInt("9223372036854775807"))
18  }
```

After:

```
1   sealed abstract class Int64 extends SignedNumber {
2     override def apply: BigInt => Int64 = bigInt => {
3       require(
4             bigInt >= BigInt("-9223372036854775808")
5         && bigInt <= BigInt("9223372036854775807"))
6
7       Int64(bigInt)
8     }
9   }
10
11  case object Int64 extends BaseNumbers[Int64] {
12    lazy val zero = Int64(0)
13    lazy val one = Int64(1)
14
15    lazy val min = Int64(BigInt("-9223372036854775808"))
16    lazy val max = Int64(BigInt("9223372036854775807"))
17
18    def apply(bigInt: BigInt): Int64 = {
19      require(
20           bigInt >= BigInt("-9223372036854775808")
21        && bigInt <= BigInt("9223372036854775807"))
22
23      Int64Impl(bigInt)
24    }
25  }
26
27  private case class Int64Impl(underlying: BigInt) extends Int64 {
28    require(underlying >= BigInt("-9223372036854775808"))
29    require(underlying <= BigInt("9223372036854775807"))
30  }
```