Enforcing Determinism of Java Smart Contracts

Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy
fausto.spoto@univr.it

Abstract. Java is a high-level, well-known and powerful object-oriented language, with a large support library and a comfortable toolbelt. Hence, it is progressively being proposed for writing smart contracts in blockchain. However, its support library is non-deterministic, which is a blocking issue for its application to smart contracts. This paper discusses the kind of (non-)determinism of the methods of the Java library and how a deterministic fragment of that library can be specified. It shows that some relevant parts are deterministic only under specific conditions on runtime values. It concludes with the description of an instrumentation, for the Takamaka blockchain, that enforces such conditions, statically or dynamically, reporting experiments with its implementation.

1 Introduction

Smart contracts are programs that specify the effects of running blockchain transactions. They are written in specialized languages, that take into account the fact that they operate on data kept in blockchain. They must support some special concepts, such as the access to the caller of a transaction, monetary transfers between contracts and payment of code execution through *gas*. Such concepts are not natively available in traditional programming languages. Instead, the Bitcoin bytecode [7, 16] can be seen as a language for smart contracts, although non-Turing equivalent and mostly limited to coin transfers. The more powerful Solidity [8], compiled into Ethereum bytecode, allows one to code complex smart contracts, in an imperative high-level language, and is one of the main reasons behind the success of Ethereum.

Recently, there have been efforts towards the use of traditional high-level languages for writing smart contracts. A notable example is Hyperledger Fabric [6, 20], that allows one to write smart contracts in Java, among other languages. Quoting from [6], "blockchain domain-specific languages are difficult to design for the implementer and require additional learning by the programmer. Writing smart contracts in a general-purpose language (*e.g.*, Go, Java, C/C++) instead appears more attractive and accelerates the adoption of blockchain solutions". In particular, Java is a well-known programming language, with modern features such as generics, inner classes, lambda expressions and lazy streams. For instance, a Solidity contract for a Ponzi pyramid scheme consists of 39 non-blank non-comment lines of code (page 155 of [13]), while its translation into

Java is only 19 lines long. Java has a large and powerful toolbelt and an active community. Also other blockchains [1, 2, 4, 3, 19] let programmers code in a limited subset of Java. It is true that Solidity has native features for smart contracts, such as `payable` functions, but the same can be obtained in Java through instrumentation [19].

One of the most compelling reasons for writing smart contracts in a mainstream language such as Java is that it comes with a large support library, that provides general solutions to typical programming problems. Many programmers are familiar with that library and appreciate the possibility of using it also for developing smart contracts. This reduces development time and errors, since the library has been widely tested in the last decades and its semantics is well-known. There is, however, a big issue here. Namely, the support library of Java is non-deterministic, in general. For most blockchains, non-determinism leads to a fork of the network, since consensus cannot be reached. Hyperledger Fabric allows instead non-determinism, in the sense that code execution, if non-deterministic, gets rejected [20]. Hence, also in this case, programmers should avoid non-determinism, if they want their code to be run. Non-determinism can be allowed in smart contracts for the generation of random numbers [11], but that technique does not apply to other forms of non-determinism.

Non-determinism is obvious for library classes and methods that perform, explicitly, parallel or random computations. But the real problem is that also some library parts, that are explicitly sequential, might lead to non-deterministic results. This is well-known to Java programmers and has been at the origin of subtle bugs also in traditional software. For smart contracts, however, this situation cannot be tolerated: the execution of the same code, from the same state, *must* lead to the same result in any two distinct blockchain nodes.

A solution could be to write a special Java library, whose methods are made deterministic. But this is far from simple, since the Java library is huge and determinism would require to modify very low-level aspects of the same virtual machine, such as its memory allocation strategy and its garbage collectors. The process should be repeated at each new version of Java. Moreover, programmers should be aware to use a non-standard library, with a different semantics. Finally, one should be sure to have fixed *all* possible sources of non-determinism in the (immense) Java library. Instead, this paper advocates the use of a standard, fixed Java library, restricted to a white-listed fragment and with the addition of a verification layer that enforces some run-time conditions. It must be ensured that this fragment is deterministic, which is typically small (but still large if compared to, say, Solidity, that has *no* native support library and only very rudimentary third-party libraries; in particular, Solidity's `library` keyword only allows one to define stateless libraries). One needn't prove to have caught every single potential case of non-determinism in the library, but only that the white-listed sandbox is safe.

Namely, the contributions of this paper are a discussion of the (non-)determinism of the Java library methods; a way for specifying deterministic fragments of that library; a technique for enforcing some run-time conditions needed for de-

terminism, statically or dynamically; an implementation of the technique, that is currently part of the verification layer for Java smart contracts of the Takamaka blockchain; experiments with that implementation. Its deterministic fragment has been specified manually, since there is currently no automatic way of proving that a method is deterministic. The Takamaka blockchain verifies and then instruments the code of the smart contracts, that gets then installed, in instrumented form, in blockchain [19]. Later, transactions execute the instrumented code, never the original one. All nodes perform verification and instrumentation and must agree on the result. An attacker cannot modify the instrumented program or its execution semantics after it has been installed in blockchain. The motivation of the original instrumentation goes beyond the issue on determinism; we refer to [19] for its description, while this paper describes its use for enforcing determinism only.

This work is organized as follows. Sec. 2 discusses the (non-)determinism of some examples of Java library methods. Sec. 3 defines the notion of *deterministic white-listed fragment* of the Java library. Its definition can use run-time conditions on the way its methods are called. Sec. 4 provides a technique for enforcing such conditions, dynamically or statically, through bytecode instrumentation. Sec. 5 reports experiments with an implementation.

This paper is *not* about Java smart contracts themselves. The interested reader is referred to the references provided above and, in particular, to [19] and to the tutorial about Takamaka smart contracts in Java[1].

2 Determinism for Java Methods in Smart Contracts

This section discusses the different kinds of determinism of methods from the Java library and what must be required if they are used in a language for smart contracts. We recall that the Java library exists in different versions, from its first 1.0 edition of 1996 to its 13 edition of September 2019. Each version has different implementations, with OpenJDK probably being the most used, nowadays. Some classes and methods exist only in some versions of the library. For instance, class `java.lang.Integer` exists from the very first 1.0 edition, while class `java.util.Collection` was only introduced with version 1.2.

Let us discuss the meaning of *determinism*. The *state* σ of the Java interpreter is typically defined as a function from variables and object fields into values. Then the goal is to guarantee that the execution ϕ of a piece of code, from the same state σ but in two different blockchain nodes, yields the same result, *i.e.*, same next state σ' and same value r (if the code is an expression): $\phi(\sigma) = \langle \sigma', r \rangle$ in all nodes of the network. This definition, however, is too strict. Consider the constructor `String()` of `java.lang.String`, that instantiates and returns a new empty `String` object. It is an expression whose execution, from any given state σ but in two different blockchain nodes, will very likely yield two distinct pointers r in RAM: the heap allocation will likely pick two distinct free locations. But both

[1] Available at `https://www.takamaka.dev/docs/Takamaka.html`

Method	Deterministic?			
	always	platform	platform + conditions	never
`Number.intValue()`	✓			
`Object()`	✓			
`String()`	✓			
`HashSet<E>()`	✓			
`String(String original)`	✓			
`String.concat(String other)`	✓			
`String.length()`	✓			
`Integer.valueOf(int i)`		✓		
`Object.hashCode()`			✓	
`Object.toString()`			✓	
`Collection<E>.iterator()`			✓	
`Collection<E>.stream()`			✓	
`Collection<E>.add(E e)`			✓	
`StreamSupport.stream(Spliterator<T> s, boolean p)`			✓	
`System.currentTimeMillis()`				✓
`BaseStream<T,S>.parallel()`				✓
`Thread.start()`				✓
	potentially white-listed			

Fig. 1. Some methods of the Java library, with their behavior *wrt.* determinism. *Always* means that a method does not compromise code determinism. *Platform* means that a method does not compromise determinism, but only if a specific implementation of the library is fixed. *Platform + conditions* means that a method does not compromise determinism, but only if an implementation of the library is fixed *and* the program satisfies extra conditions, that typically refer to the actual arguments passed to that method or to other methods at run time. *Never* means that a method can have different behaviors in distinct executions, even on a specific library implementation, and no condition can be sensibly devised to make its behavior deterministic.

references will refer to a brand new empty string. As long as the programming language does not allow one to distinguish the exact pointer, but only the object it refers to, then such two next states and results can be considered *equivalent*. This notion of state and reference equivalence is actually borrowed from [9, 10]. With this interpretation in mind, `String()`, applied to two equivalent states but in two distinct blockchain nodes, yields equivalent states and equivalent results (two references to brand new empty strings), and is consequently deterministic.

Fig. 1 reports some examples of methods from the Java library, classified on the basis of the kind of determinism that holds for them. We discuss them below. We recall that, in Java, a method call `o.m(pars)` specifies its *receiver* `o` and its *static target* `C.m(types)`, that is, a method signature reporting the class `C` from where method `m` with formal arguments of type `types` must be looked for. Note that `C` and `types` do not include generic type parameters, if any, since they are

erased during compilation from Java to Java bytecode and our instrumentation works at bytecode level. Since Java is an object-oriented language, the static target is just the specification of the method implementation that must be run (the *dynamic* target): a non-static method call runs the implementation of `C.m(types)` that is selected by looking up `m(types)` from the run-time class of `o` upwards, along the superclass chain.[2] Hence, any implementation of `m(types)` in `C` or in any of its subtypes can be run. For instance, a call `o.intValue()` with static target `Number.intValue()` can be called on any object `o` that extends `java.lang.Number`. At run time, it might execute any implementation of `intValue()` in any subtype of `Number`, such as in `Integer` or in `java.lang.Double`, depending on the run-time type of `o`. Hence, when Fig. 1 classifies `Number.intValue()` as always deterministic, this applies to every implementation of `intValue()` in the subtypes of `Number` of the Java library.

Many methods of the Java library are clearly deterministic. Their behavior is fixed by the official documentation by Oracle and does not change across distinct versions of the library. An example is the unwrapping method `intValue()` of `Number` (Fig. 1). It yields the primitive `int` value corresponding to an instance of the abstract class `Number`, such as objects of class `Integer` or `Double`. For instance, `new Integer(3).intValue() == 3` holds in any version of the Java library, as well as `new Double(3.14).intValue() == 3` since, in Java, truncation of `double` into `int` is machine-independent. Hence, a language for smart contracts, that must require determinism, can safely allow the use of that method, always. Other examples are the constructors of `String`, `java.lang.Object` and `java.util.HashSet` reported in Fig. 1, or methods `String.concat(String other)`, that yields the concatenation of strings `this` and `other`, and `String.length()`, that yields the length of a string. If only these methods are used, then any Java library can be used by any node of a blockchain, even distint versions in distinct nodes.

The static method `Integer.valueOf(int i)` wraps a primitive `int` value into an object of class `Integer`. It might be surprising, but its result can be different in two blockchain nodes, if they use distinct implementations of the Java library. For instance, while `Integer.valueOf(3) == Integer.valueOf(3)` holds in every implementation of the library, since the official documentation requires this method to cache values between -128 and 127, inclusive, there is instead no guarantee that caching is used outside that range. Hence `Integer.valueOf(2019) == Integer.valueOf(2019)` might be true in some implementations of the Java library and false in others (such as in Oracle JDK 13.0.1). We call *platform deterministic* such methods, since they are deterministic only once a specific implementation (*platform*) of the Java library is fixed. If only methods of this and of the previous category are used, then all nodes of a blockchain must run on a given, fixed Java library, to guarantee determinism.

[2] Java also allows so-called *special* calls, such as `super.m()`, that start the look-up from a given *static* type; as well as calls embedded in closures, such as method references (corresponding to `invokedynamic` in Java bytecode). For simplicity, these calls are not discussed here, but our implementation deals with them.

Consider method `Object.hashCode()` of class `Object` now. It yields an `int` hash of its receiver. Its implementation computes that hash from the RAM pointer value of the object reference. Since two blockchain nodes will likely use different RAM pointers, this method is non-deterministic. In other terms, this method exposes an execution detail (the exact RAM pointer) that was meant to be invisible for state equivalence. For instance, `int i = new Object().hashCode()` will likely assign different values to `i` in two distinct blockchain nodes (and even in repeated executions on the same node). The same problem occurs for `Object.toString()` that, inside `Object`, is implemented in terms of `Object.hashCode()` (its implementation concatenates the name of the run-time class of its receiver with its hash and returns it). Hence, `String s = new Object().toString()`, in two distinct blockchain nodes, will likely assign different strings to variable `s`. However, banning calls to `Object.hashCode()` or `Object.toString()` from smart contracts would be unacceptable to programmers, that heavily use such calls in their programs, without incurring in non-determinism. The reason is that programmers normally take care of calling such methods only on objects that redefine the default (non-deterministic) implementation of `hashCode()` and `toString()` in class `Object`. If that is the case, the calls will actually execute the deterministic redefinitions. Hence, it seems sensible to allow calls to such methods in smart contracts, but only if their receiver redefines them in a deterministic way, as in: `String o = ...; int h = o.hashCode()`, where `o` holds a `String`, that redefines `hashCode()`. Hence, such methods are *platform-deterministic under certain conditions*: they are deterministic if a given Java library is fixed *and* some run-time conditions hold. Sec. 4 shows how such conditions can be enforced.

Consider methods `iterator()` and `stream()` of the generic `Collection<E>`. They provide two ways for processing the elements, of type E, of a collection. The former implements the traditional *iterator pattern* and yields an object that enumerates the elements. The latter yields a *stream*, *i.e.*, a lazy algorithm on the elements, that can be subsequently programmed and executed. Streams implement the map/filter/reduce pattern, making heavy use of lamba expressions. Interestingly, neither method guarantees a fixed enumeration order. There are collections for which they guarantee an order, such as instances of `java.util.List<E>`: on lists, enumeration proceeds from head to tail; or instances of `java.util.LinkedHashSet<E>`, on which they proceed in insertion order. For collections such as `java.util.HashSet<E>`, instead, the order varies with the library version *and* at each execution. The reason is that `HashSet` uses a hashmap [12] to store elements with the same `hashCode()` in the same bucket. Since `hashCode()`, as shown above, is non-deterministic, then the distribution of elements in the buckets varies from run to run and their enumeration as well, being the lexicographic scan of the buckets' elements. One could forbid `HashSet` and only allow its more expensive sibling `LinkedHashSet`, whose iteration order is fixed. But then method `add()` would still be non-deterministic, since its gas consumption is affected by the shape of the buckets, as discussed later. Moreover, programmers use `HashSet` extensively and would be annoyed if it were banned. It is much better to observe that, if the `hashCode()` of all its elements has been

redefined in a deterministic way and if a specific library version is fixed, then the behavior of `HashSet` becomes deterministic, since the shape of the buckets is the same in every run. This means that one can allow, in smart contracts, calls to `iterator()` and `stream()` on any collection, but only under such conditions. Sec. 4 shows how they can be enforced.

Consider method `add(E e)` of `Collection<E>`. It adds an element `e` to the collection. For a given library version, it is deterministic on lists: it adds `e` to the end of the list. However, on `HashSet` and `LinkedHashSet` it scans the bucket selected for `e.hashCode()` and checks if an equal element was already in that bucket, by calling `equals()` against each of its elements. Hence `add(E e)` on a hashset consumes an amount of gas that depends on the shape of its buckets. Again, the solution is to require that all elements of the set and `e` redefine `hashCode()` in a deterministic way, which must be enforced at run time (Sec. 4).

Consider the static method `stream(Spliterator<T> s, boolean p)` of class `java.util.stream.StreamSupport`. It yields a stream for processing the elements specified by the given `Spliterator`. If the `Spliterator` is deterministic, the resulting stream is deterministic as well, on a given library version, but only if it is sequential. Passing true for `p` would yield a parallel stream instead, that is inherently non-deterministic. Hence, this method can be used only if a specific library is used and if it is enforced that false is passed for `p` at run time (Sec. 4).

If only methods of this and of the previous two categories are used, then all nodes of a blockchain must be run on a given, fixed Java library, *and* the run-time conditions that entail determinism must be somehow enforced.

Static method `System.currentTimeMillis()`, in `java.lang.System`, yields the number of milliseconds elapsed since the beginning of 1970. Not surprisingly, it will yield different measures in different blockchain nodes. Such an inherently non-deterministic method cannot be used in a smart contract. Consider method `parallel()` of `java.util.stream.BaseStream` now. It yields a parallel version of a stream. For instance, if `list` is a list with at least two distinct positive `Integer`s:

```
int positive = list.stream().parallel().mapToInt(Integer::intValue)
  .filter(i -> i > 0).findAny().getAsInt();
```

processes `list` with a parallel algorithm that unwraps each `Integer` element of the list into its corresponding `int` primitive value, filters only the positive values and selects any of them. The result of `findAny()` is an optional value, hence the `getAsInt()` call at the end. Since the algorithm is parallel, each execution of this code might select a different positive element, depending on thread scheduling. Hence, `parallel()` introduces non-determinism and cannot be allowed in smart contracts. Note that even the gas consumption of the code is not deterministic, since it depends on how many elements are checked before a thread encounters a positive value and terminates the look-up.

In general, methods that introduce parallelism are never deterministic. Another example is `Thread.start()` in `java.lang.Thread`, that spawns a parallel execution thread. If methods of this category are used, then the code cannot be used in blockchain, since there is not way to make it deterministic.

3 White-Listed Fragments of the Java Library

Sec. 2 has shown that some Java library methods can be used in smart contracts, at least if a specific version of the library is fixed and some run-time conditions are enforced. Such methods can hence be *white-listed* for smart contracts (Fig. 1).

Definition 1. *A* white-listed fragment *WL is a set of method signatures (constructors are considered methods named as their defining class), with associated run-time conditions (if any).*

The consensus rules must specify a white-listed fragment WL: each node verifies that smart contracts *obey* WL, or otherwise aborts their execution in blockchain.

Definition 2. *A program P* obeys *a white-listed fragment WL if and only if*

1. *every method call in P has a static target that is either in contract code, or in WL or overrides a signature in WL (syntactical check); and*
2. *in every execution of P, the run-time conditions in WL hold (semantical check).*

A program P that obeys a white-listed fragment WL cannot call signatures outside WL, but can call signatures in WL that, indirectly, call methods outside WL. In other terms, Def. 2 constrains only the library API allowed in P.

Definition 3. *Given a Java library version V, a white-listed fragment WL is* deterministic for *V if and only if any Java program that obeys WL is deterministic, when executed over V.*

A very simple example of white-listed fragment is $WL_1 = \emptyset$. It is deterministic for every Java library version. Namely, a program P that obeys WL_1 cannot call any library method nor constructor (condition 1 of Def. 2). P cannot contain classes, since Java classes have always at least a constructor that calls the constructor of `Object`, possibly indirectly. Hence, P consists of interfaces only, with default and static methods that do not call any library code, and is hence deterministic. A deterministic white-listed fragment is $WL_2 = \{$`Object()`$\}$. This time, one can write programs P that obey WL_2, with classes that extend `Object` and whose constructors call `Object()`. But no other library methods nor contructors can be called, which is an irrealistic constraint. The white-listed fragment $WL_3 = \{$`Object()`, `System.currentTimeMillis()`$\}$ is not deterministic, since it is possible to write a program, that obeys WL_3, consisting of a single class with a method that uses `System.currentTimeMillis()` to return a random value. A deterministic white-listed fragment that allows, at least, simple string manipulations is $WL_4 = \{$`Object()`, `String.concat(String other)`, `String.length()`$\}$. It allows one to write Java programs whose classes extend `Object` and whose code performs computations such as `"hello".concat(s).length()`, where `s` is a string.

Up to now, we have not used run-time conditions in white-listed fragments. In order to specify such conditions, it is possible to use Java annotations on method signatures. An annotation can be applied to a formal parameter, meaning that

the condition must hold for the corresponding actual parameter; or to a non-static method itself, meaning that the condition must hold for the receiver of the method. Let us introduce for instance the following annotations:

`@MustRedefineHashCode`: the annotated value is `null` or belongs to a class that redefines `Object.hashCode()`;
`@MustRedefineHashCodeOrToString`: the annotated value is `null` or belongs to a class that redefines `Object.hashCode()` or `Object.toString()` (or both);
`@MustBeFalse`: the annotated value is `false`.

Then one can define

$$WL_5 = \left\{ \begin{array}{c} \texttt{Object()}, \\ \texttt{@MustRedefineHashCodeOrToString Object.toString()}, \\ \texttt{HashSet<E>()}, \\ \texttt{Collection<E>.add(@MustRedefineHashCode E e)} \end{array} \right\}$$

that constrains `Object.toString()` to be called on values that redefine at least one of `hashCode()` and `toString()`, and `Collection<E>.add(E e)` to be called with an actual parameter that redefines `hashCode()`. WL_5 is deterministic for any given Java library version, thanks to such constraints. For instance, the following code is deterministic on any Java library:

```
Set<Object> set = new HashSet<>();
set.add("hello"); set.add(BigInteger.ONE); set.add(new HashSet<String>());
String s = set.toString();
```

It creates a hashset and populates it with a `String`, a `java.math.BigInteger` and an empty `HashSet`. These redefine `hashCode()`, hence the constraint on `add()` holds. At the end, it calls `toString()` on the `set`, that is a `HashSet` that redefines `toString()`, hence the constraint on `toString()` holds. By using, for instance, the OpenJDK 13 library, variable `s` will always hold the string `"[[], 1, hello]"`. Other versions of the library might compute different strings but, once a library version is fixed, always the same string is computed. The reason of this determinism is that the shape of the buckets of the hashset is fixed since the `hashCode()` of its elements is redefined in a deterministic way in `String`, `BigInteger` and `HashSet`. Moreover, `toString` on `HashSet` has been redefined in a way that iterates on the elements of the hashset and concatenates their `toString()`, which is redefined and deterministic in `String`, `BigInteger` and `HashSet`.

The justification above for the determinism of the code follows from a manual investigation of the library's source code. It is not automated. What can be automated is, instead, the verification that a program P obeys a given WL. Namely, condition 1 of Def. 2 can be verified by following the static target of each call in P, upwards, and checking if the method is in WL. Condition 2 of Def. 2 is more complex and Sec. 4 shows how it can be enforced.

A white-listed fragment WL must be specified in a way that is easily machine-readable. WL is a set of method signatures, hence it can be provided as a set of Java abstract classes and interfaces. For each library class `C` that defines

some methods to white-list, one writes an abstract class `whitelisted.C`; for each interface `I`, one write an interface `whitelisted.I`. That is, one defines, in package `whitelisted`, *mirrors* declaring the white-listed signatures and their annotations, if any. The advantage of using such mirrors is that they can be written by copying and pasting signatures from the source code of the mirrored library classes. Moreover, they can be compiled, which helps spotting typos. Finally, a blockchain node can query such classes by reflection, which is both simple and efficient, compared for instance to querying textual or XML specifications.

For instance, Fig. 2 specifies a white-listed fragment of `Object`, `Collection`, `Set`, `List`, `HashSet` and `ArrayList`[3]. It is deterministic for a given fixed version of the Java library: thanks to its annotations, this fragment guarantees that elements in collections (hence also in sets and lists) redefine `hashCode()`, which makes iterators on collections, streams derived from collections and `toString()` on collections deterministic. The latter can be called on collections since they redefine it, hence satisfying the constraint for `Object`.

It is interesting to observe that method `contains(Object o)` in `Collection` (Fig. 2) requires `o` to redefine `hashCode()` or otherwise, for hashsets, the bucket where `o` is looked for might be different for different runs, with subsequent non-determinism. That constraint has been lifted for `contains(Object o)` in `List` (Fig. 2), since its implementation, for lists, scans the list from head to tail, looking for an `equals()` element, with deterministic behavior, regardless of `o` having redefined `hashCode()` or not. This is an application of the Liskov substitution principle [14]: remember that that principle, for formal arguments, works by generalization: hence white-listing constraints on formal arguments can be weakened in subclasses, but not strengthened. The same for `remove(Object o)`, passing from `Collection` to `List`.

There is no single largest deterministic fragment for a given set of classes and interfaces. For instance, another deterministic fragment for the same classes in Fig. 2 is identical but for allowing any elements in lists, possibly not redefining `hashCode()` (hence more permissive); forbidding `Object.toString()` and `Object.hashCode()` altogether, since it would now be non-deterministic on lists (hence more restrictive); and forbidding the constructor `HashSet(Collection<? extends E> c)` (Fig. 2), that allows a hashset to be built from a list and hence contain elements whose `hashCode()` has not been redefined, with consequent non-determinism (hence more restrictive). Choosing a specific deterministic fragment is often a question of personal taste. One should choose that allowing more methods of frequent use. In this example, forbidding `Object.toString()` and `Object.hashCode()` would hardly be acceptable.

4 Enforcing Run-Time Conditions for Determinism

Sec. 3 shows that a deterministic fragment of the Java library can require run-time conditions on the values of the receiver or parameters of its methods. A

[3] These signatures are copy and paste from the library source code. For the use of generics, wildcards and `Object` in these signatures, we refer to [15].

```
public abstract class whitelisted.java.lang.Object {
  public Object() {}
  public abstract boolean equals(java.lang.Object other);
  public abstract @MustRedefineHashCodeOrToString java.lang.String toString();
  public abstract @MustRedefineHashCode int hashCode();
}

public interface whitelisted.java.util.Collection<E> {
  int size();
  boolean isEmpty();
  java.lang.Object[] toArray();
  <T> T[] toArray(T[] a);
  <T> T[] toArray(java.util.function.IntFunction<T[]> generator);
  boolean add(@MustRedefineHashCode E e);
  boolean contains(@MustRedefineHashCode java.lang.Object o);
  boolean remove(@MustRedefineHashCode java.lang.Object o);
  boolean containsAll(java.util.Collection<?> c);
  boolean addAll(java.util.Collection<? extends E> c);
  boolean removeAll(java.util.Collection<?> c);
  boolean removeIf(java.util.function.Predicate<? super E> filter);
  boolean retainAll(java.util.Collection<?> c);
  void clear();
  java.util.stream.Stream<E> stream();
  java.lang.Iterator<E> iterator();
}

public interface whitelisted.java.util.Set<E> {
  boolean containsAll(java.util.Collection<?> c);
  boolean addAll(java.util.Collection<? extends E> c);
  boolean retainAll(java.util.Collection<?> c);
  boolean removeAll(java.util.Collection<?> c);
}

public interface whitelisted.java.util.List<E> {
  E get(int index);
  E remove(int index);
  boolean remove(java.lang.Object o);
  boolean contains(java.lang.Object o);
  void sort(java.util.Comparator<? super E> c);
  E set(int index, @MustRedefineHashCode E element);
  void add(int index, @MustRedefineHashCode E element);
  int indexOf(java.lang.Object o);
  int lastIndexOf(java.lang.Object o);
  java.util.ListIterator<E> listIterator();
  java.util.ListIterator<E> listIterator(int index);
  java.util.List<E> subList(int fromIndex, int toIndex);
  java.util.Spliterator<E> spliterator();
}

public abstract class whitelisted.java.util.HashSet<E> {
  public HashSet() {}
  public HashSet(java.util.Collection<? extends E> c) {}
  public HashSet(int initialCapacity, float loadFactor) {}
  public HashSet(int initialCapacity) {}
}

public abstract class whitelisted.java.util.ArrayList<E> {
  public ArrayList() {}
  public ArrayList(int size) {}
  public ArrayList(java.util.Collection<? extends E> c) {}
  public abstract void trimToSize();
  public abstract void ensureCapacity(int minCapacity);
}
```

Fig. 2. White-listed methods of `Object`, `Collection`, `Set`, `List`, `HashSet` and `ArrayList`.

blockchain node must enforce that such conditions hold at run time for the smart contracts that it executes. Hence each condition is a proof-obligation that must be discharged: if this is not possible, the smart contract cannot be executed or, at least, its execution must be aborted. This section shows how this is possible.

Very likely, a blockchain node receives the smart contract already compiled in Java bytecode. The idea is hence to let the node instrument such bytecode, only the first time it is installed in blockchain, with extra checks that, at each subsequent execution, verify the run-time conditions. To make the technique more accessible to the reader, this section presents the instrumentation at source-code level, but it actually works at bytecode level.

Assume that a blockchain node verifies that smart contracts obey to the deterministic fragment in Fig. 2. Assume that a user installs in blockchain a smart contract whose code contains `collection.remove(element)`, whose static target is `Collection<E>.remove(Object o)`. The node spots this syntactically.[4] The node consults its white-listed fragment and recognizes the call as white-listed, but having a run-time constraint `@MustRedefineHashCode` on `o` (Fig. 2). Hence, during installation of the smart contract, the node instruments its code by adding a brand new verification method:

```
private static boolean remove_0(Collection<E> receiver, Object par_0) {
  Support.mustRedefineHashCode(par_0);
  return receiver.remove(par_0);
}
```

and replaces `collection.remove(element)` with `remove_0(collection, element)`. Each time that code will later be executed, `Support.mustRedefineHashCode(par_0)` will check (through Java reflection) that the actual argument to `remove(Object o)` redefines `hashCode()` and aborts the current transaction otherwise. The node includes a `Support` class for that, whose code is not reported for space limitations.

For another example, assume that the smart contract calls `x.toString()`, with static target `Object.toString()`. The blockchain node spots this syntactically, consults its white-listed fragment and recognizes the call as white-listed, but having a run-time constraint `@MustRedefineHashCodeOrToString` on `x` (Fig. 2). Consequently, it replaces `x.toString()` with `toString_0(x)` and adds the verification method:

```
private static toString_0(Object receiver) {
  Support.mustRedefineHashCodeOrToString(receiver);
  return receiver.toString();
}
```

At each run of the contract, `Support.mustRedefineHashCodeOrToString(receiver)` will be executed later, that checks the condition by reflection.

Assume that the smart contract contains a static call `StreamSupport.stream(s, p)`, whose static target is `StreamSupport.stream(Spliterator<T> s, boolean p)`,

[4] The Java bytecode of the smart contract will contain an instruction `invokeinterface java.util.Collection.remove(Object):boolean`, or a similar one for a subtype of `Collection`.

and that the white-listed fragment of the blockchain node allows that signature, but has a run-time condition on `p` that avoids the creation of parallel streams: `StreamSupport.stream(Spliterator<T> s, @MustBeFalse boolean p)`. The node replaces `StreamSupport.stream(s, p)` with `stream_0(s, p)` and adds the verification method:

```
private static Stream<T> stream_0(Spliterator<T> par_0, boolean par_1) {
  Support.mustBeFalse(par_1);
  return StreamSupport.stream(par_0, par_1);
}
```

`Support.mustBeFalse(value)` aborts the current transaction when `value` is false.

4.1 Static vs. Dynamic

The instrumentation technique described above adds dynamic checks on run-time values, that will be triggered during each subsequent transaction. Hence, checks are performed repeatedly, every time an annotated white-listed method is executed. This can incur in a performance penalty. It would be better to check, once and for all, if a run-time condition holds, definitely, when smart contracts are installed in blockchain. This can be done with static analysis [17], a technique that infers properties of programs, before they are actually run. Since the verification of non-trivial run-time program properties is in general undecidable [18], static analysis provides a definite answer only in some cases. Hence, a blockchain node can use static analysis to discharge the proof-obligations due to run-time conditions on white-listed methods. If it succeeds with a definite answer, stating that a given condition definitely holds, the node needn't generate any verification method for that condition. Otherwise, it adds the verification method, as last resort.

Static analysis can be more or less aggressive. More aggressive static analyses discharge more proof-obligations statically, which is desirable since the smart contract's code will check less conditions at run-time. However, aggressive analyses are typically more expensive (although they are executed only once, when the smart contract is installed in blockchain). In practice, a good trade-off should be found between the power of the analysis and its cost.

Our implementation uses static type information to infer, statically, if a condition `@MustRedefineHashCode` or `@MustRedefineHashCodeOrToString` holds for a variable `v`. If the static type τ of `v` is a class that redefines `Object.hashCode()` or `Object.toString()`, the same must hold for the dynamic type τ' of `v`, that can only be an instance of τ. Hence the run-time condition must hold, always. This follows from the fact that Java and Java bytecode are strongly-typed. Otherwise, the verification method is added for that condition. For `@MustBeFalse`, our implementation looks, intra-procedurally, for the producers of the annotated value. If these are always the literal `false`, then the condition holds. If, instead, at least one producer is the literal `true` or a complex expression, then the static analysis gives up and the verification method is added.

archive of smart contracts	#m	#c	#d	without	with	instr. time
auctions.jar	20	0	0	11997	11997	4
basicdependencies.jar	38	3	0	7265	7505	26
basic.jar	38	8	0	8109	8464	31
collections.jar	0	0	0	8909	8909	4
crowdfunding.jar	0	0	0	10043	10043	4
io-takamaka-code.jar	348	18	5	88101	89177	223
javacollections.jar	38	17	0	3040	3635	18
lambdas.jar	45	3	0	4570	4745	27
ponzi.jar	69	7	0	12964	13556	43
purchase.jar	0	0	0	5362	5362	4
tictactoe.jar	40	0	0	6745	6745	27
voting.jar	8	1	0	7210	7305	20

Fig. 3. The jars used for the experiments. Each contains one or more contracts. For each jar, #m is the number of calls to signatures of the Java library; #c is the number of white-listing conditions that must be checked for those calls; #d is the subset of #c that has been discharged statically, at instrumentation time; *without* is the size (in bytes) of the instrumented jar without the addition of run-time checks for the remaining $\#c-\#d$ conditions; *with* is the size (in bytes) of the instrumented jar with that addition; *instr. time* is the time for computing such instrumented jars (in milliseconds).

5 Experiments

We have implemented the technique of Sec. 4 at bytecode level, by using the BCEL library for bytecode manipulation [5]. Experiments have been performed on an Intel 4-Core i3-4150 at 3.50GHz with 16GB of RAM, running Linux Ubuntu 18.04.

Fig. 3 reports the Java archives (jars) used for the experiments. They contain contracts for auctions (`auctions.jar`), for storage of objects in blockchain (`basicdependencies.jar`), for the use of such objects as a library (`basic.jar`), a library of collection classes explicitly developed for Takamaka (`collections.jar`), contracts for crowdfunding (`crowdfunding.jar`), the same run-time support library of Takamaka (`io-takamaka-code.jar`), contracts using the Java collections (`javacollections.jar`), contracts testing lambda expressions (`lambdas.jar`), Ponzi contracts (`ponzi.jar`), a remote purchase contract (`purchase.jar`), a tic-tac-toe game contract (`tictactoe.jar`) and contracts for electronic voting (`voting.jar`).

Column #m of Fig. 3 reports how many calls to constructors or methods of the Java library occur in the code. Our implementation uses an extension of the fragment in Fig. 2 and verified that all those calls are white-listed. However, a few of them require to enforce run-time conditions for determinism (Sec. 4); namely, column #c counts such conditions; of these, our implementation could statically discharge #d, at instrumentation time, through static analysis; that is, these needn't be checked at run time, since these checks are eliminated by the simple static analysis of Sec. 4.1. The remaining $\#c - \#d$ require the addition

of run-time checks in the instrumented jar. Consequently, when $\#c - \#d > 0$, extra code is instrumented into the jars, as it can be seen by comparing columns *without* and *with* in Fig. 3. These show that the extra checks induce a very small inflation of the instrumented jars. Finally, Fig. 3 reports the time for instrumentation, in milliseconds. We computed it for columns *with* and *without*, but the results (reported in the figure) were identical. This means that the time for checking if method calls are white-listed and for adding run-time checks for white-listing conditions is very small. In order to investigate if the extra run-time checks for the $\#c - \#d$ conditions slow down the code, we have run 113 JUnit tests, that trigger blockchain transactions that execute the instrumented code of the jars in Fig. 3 (these tests are in the companion archive). The total time for running the tests (115 seconds) was the same both with and without the addition of the run-time checks, which shows that these do not actually slow down the execution. Note that the code without run-time checks is still instrumentated for running on a blockchain. We cannot compare with fully uninstrumented code since it cannot be run [19].

6 Conclusion

The technique in this paper allows a simple specification of a deterministic fragment of the Java library and enforces its run-time constraints. Experiments show that it works in practice and does not incur in size or time degradation of the compiled code. Current work consists in enlarging the white-listed fragment. We will perform this task on demand, while writing smart contracts in Takamaka, in order to concentrate only on library portions that are relevant for that.

References

1. Aion Foundation. `https://aion.network`.
2. Hello World...from the Aion Virtual Machine! `https://blog.aion.network/hello-world-from-the-aion-virtual-machine-25038ac62f17`.
3. How to Use Java to Write a NEO Smart Contract. `https://docs.neo.org/docs/en-us/sc/devenv/getting-started-java.html`.
4. NEO - An Open Network for Smart Economy. `https://neo.org`.
5. BCEL. `https://commons.apache.org/proper/commons-bcel`, December 2017.
6. E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolic, S. Weed Cocco, and J. Yellick. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proc. of the Thirteenth EuroSys Conference, EuroSys 2018*, pages 30:1–30:15, Porto, Portugal, 2018. ACM.
7. A. M. Antonopoulos. *Mastering Bitcoin: Programming the Open Blockchain*. Oreilly & Associates Inc, 2nd edition, June 2017.
8. A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps*. Oreilly & Associates Inc, 1st edition, November 2018.
9. A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

10. G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.
11. K. Chatterjee, A. K. Goharshady, and A. Pourdamghani. Probabilistic Smart Contracts: Secure Randomness on the Blockchain. In *Proc. of the IEEE International Conference on Blockchain and Cryptocurrency (ICBC'19)*, pages 403–412, Seoul, South Korea, 2019. IEEE.
12. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
13. K. Iyer and Dannen. C. *Building Games with Ethereum Smart Contracts*. Apress, 2018.
14. B. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
15. M. Naftalin and P. Wadler. *Java Generics and Collections*. O'Reilly, 2006.
16. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, October 2008.
17. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
18. H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, pages 358–366, 1953.
19. F. Spoto. A Java Framework for Smart Contracts. In *Proc. of the 3rd Workshop on Trusted Smart Contracts*, Lecture Notes in Computer Science, St. Kitts, 2019. To appear.
20. M Vukolić. Rethinking Permissioned Blockchains. In *Proc. of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts (BCC'17)*, pages 3–7, Abu Dhabi, United Arab Emirates, April 2017. ACM.