On-Chain Smart Contract Verification over Tendermint [⋆]

Luca Olivieri[1,2], Fausto Spoto[1], and Fabio Tagliaferro[1]

[1] Dipartimento di Informatica, Università di Verona, Italy
[2] Corvallis S.r.l. , Padova, Italy
`{luca.olivieri, fausto.spoto, fabio.tagliaferro}@univr.it`

Abstract. Smart contracts are computer code that runs in blockchain and expresses the rules of an agreement among parties. A bug in their code has major consequences, such as rule violations and security attacks. Smart contracts are immutable and cannot be easily replaced to patch a bug. To overcome these problems, there exist automatic static analyzers that find bugs before smart contracts are installed in blockchain. However, this *off-chain* verification is *optional*: programmers are not forced to use it. This paper defines *on-chain* verification instead, that occurs inside the same blockchain nodes, when the code of smart contracts is installed. It acts as a *mandatory* entry filter that bans code that does not abide to the verification rules, that are consequently part of the consensus rules of the blockchain. Thus, an improvement in on-chain verification entails a consensus update of the network. This paper provides an implementation of on-chain verification for smart contracts written in the Takamaka subset of Java, running as a Tendermint application. It shows that on-chain verification works, reporting actual experiments.

Keywords: Smart contract · software verification · program analysis · blockchain · Tendermint.

1 Introduction

Blockchain is a distributed ledger that replicates data in a peer-to-peer network of nodes. Transactions are ledger updates, digitally signed by the users. The nodes of the network collect broadcasted transactions into a growing cryptographically-linked chain of blocks. They execute a consensus algorithm to agree on the ledger evolution. Once consensus is achieved, it is hard, or impossible, to withdraw transactions from the blockchain. In this sense, blockchains are *immutable*. *Smart contracts* specify rules and effects of transactions and can be either built-in or given as custom code installed inside the same blockchain.

Bitcoin [9,1], in 2008, was the first popular blockchain implementation. It is a peer-to-peer electronic cash system that stores and transmits value in a currency

[⋆] Work supported by FSE – Regione del Veneto: DGR N. 1463/2019, *Innovazione e ricerca per un Veneto più competitivo – Assegni di ricerca anno 2019.*

called *bitcoin*, using a *Proof of Work* (PoW) consensus algorithm. A Turing-incomplete low-level language specifies Bitcoin's transactions. It can be seen as a limited scripting language for smart contracts. In 2013, *Ethereum* [4,2] introduced a Turing-complete bytecode for smart contracts, for developing decentralized applications. Ethereum smart contracts can be programmed in high-level languages, with Solidity being the most popular one, and run on the Ethereum virtual machine. Ethereum uses PoW but is currently switching to *Proof of Stake* (PoS), a consensus algorithm with reduced resource consumption [12]. The Tendermint protocol [8] provides a generic and customizable infrastructure for networking and consensus through PoS, with a pseudo-random election of the *validator* node for the next block. Network participants who want to become validators freeze a certain amount of *stake*, that acts as an economic incentive that dissuades from validating or creating fraudulent transactions. If the network detects a fraudulent transaction, the culprit loses part of its stake and the right to act as a validator. Tendermint's protocol tolerates up to $\frac{1}{3}$ of misbehaving nodes. Tendermint leaves the notion of transaction unspecified: programmers can develop an application layer for Tendermint, that specifies which transactions exist and which is their semantics. The application layer can be written in any programming language and can be an actual environment for the execution of Turing-complete smart contracts, similarly to Ethereum.

Not surprisingly, Turing-completeness for smart contracts introduces the risk of all sort of bugs [3,11]. Since smart contracts deal with money and cannot be replaced, it is paramount to install only *correct* code in blockchain. Thus, there exist many analyzers that verify smart contracts before they get installed in blockchain. For instance, `https://mythx.io` is an analysis service for Solidity that uses symbolic analysis to detect software vulnerabilities. Echidna [7] uses a fuzzing approach to find a sequence of transactions that violates a given property. Slither [6] uses data-flow and taint analysis to find potential issues. Furthermore, there are companies that provide code audit services, using both automatic tools and human investigation. A limit of these tools and procedures is that they are *optional* and *external* to the blockchain (hence *off-chain*): the latter does not actively protect itself against the deployment of bugged or dangerous code.

This paper makes the following contributions:

- It defines *on-chain* code verification, where the nodes of the blockchain verify the code being deployed. That is, the same network, *internally*, runs a *mandatory* code verification step and rejects code that does not pass it. As a consequence, on-chain verification is a defensive, proactive technique that guarantees that all code executed in blockchain has been successfully verified.
- It describes an actual implementation of a blockchain with on-chain verification, built as a Tendermint application that runs smart contracts written in the Takamaka subset of Java [14]. Note that we have used Tendermint as a third-party tool over which we integrate our code. Nothing has been changed in Tendermint. Our code includes 26 on-chain checks, that mostly verify the correct use of Takamaka's primitives and code annotations and the use of a deterministic subset of Java [15].

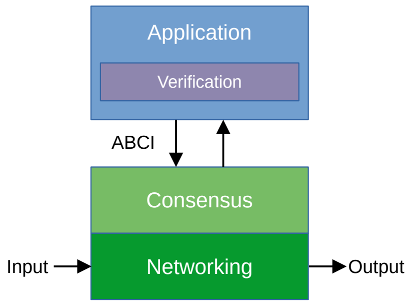

Fig. 1. High-level architecture of an application running on Tendermint Core and performing on-chain verification.

– It describes a lazy re-verification approach that copes with the evolution of the code verification rules. Namely, on-chain verification is part of code deployment transactions, hence its rules are consensus rules whose evolution requires a network update. Moreover, code previously successfully verified with old verification rules might fail to pass new verification rules. The implementation of this last contribution is not available yet.

This paper is organized as follows. Sec. 2 defines a general architecture for on-chain code verification. Sec. 3 describes our implementation of on-chain verification, over Tendermint, and shows an on-chain check. Sec. 4 reports experiments with our implementation and describes how readers and reviewers can validate them. Sec. 5 shows how the blockchain can cope with the evolution of code verification rules. Sec. 6 comments on limitations, related and future work.

2 On-Chain Code Verification

This section defines the architecture of a blockchain node with on-chain code verification, built over Tendermint. Following Fig. 1, it consists of three layers:

Networking: discovers and connects nodes with each other, propagates requests for transactions and collects their responses from other nodes.
Consensus: compares and approves/rejects the responses obtained by executing the requests on the nodes.
Application: specifies which requests are valid, how their responses are computed and how the application's state consequently evolves.

Tendermint Core is an implementation of networking and consensus, without any application layer (its distribution includes a few toy applications, irrelevant for our purposes). Programmers develop their own application layer and plug it into Tendermint Core via its Application BlockChain Interface (ABCI). Tendermint Core replicates the application state on each machine of the network.

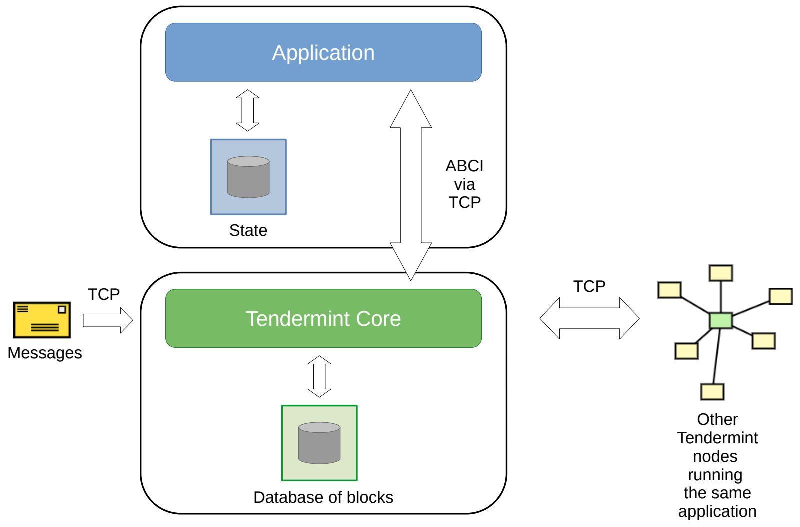

Fig. 2. Tendermint Core and a Tendermint application, with their respective databases.

Fig. 2 shows a more detailed picture of Tendermint Core and of an application connected through its ABCI. It shows that Tendermint Core keeps the blocks of the blockchain in its own database, that needn't be the same used to hold the application's *state*. The latter holds, for instance, the code of the smart contracts installed in blockchain and the value of their state variables. Tendermint Core needs only the hash of the application state, for consensus, to ensure that all nodes have reached the same application state.

One can define the application state as a map σ from the hash of the requests that the blockchain has executed to the responses that have been computed for them. The application state contains the full responses, but only the hash of the requests. Hence, it can be implemented as a Merkle-Patricia trie. The full requests are contained in the database of blocks of Tendermint Core instead, since they are needed to replay the transactions in all nodes of the network.

On-chain code verification requires a code verification module (Fig. 1). This is part of the application layer, since it contributes to the execution of the application-specific requests for code installation in blockchain. Assume that a *request*, whose hash is $request_h$, reaches the blockchain, requiring to install, in blockchain, the code of some smart contracts, reported inside *request*.

Fig. 3 shows the sequence diagram for the execution of *request*. Namely, Tendermint Core routes *request* through networking and consensus up to the application, that uses its verification module to either approve or reject the code. If approved, the application includes the code in a *response* and updates its state σ with a new binding: $\sigma(request_h) = response$. The hash $request_h$ is an immutable, machine-independent reference to this code, used later to instantiate

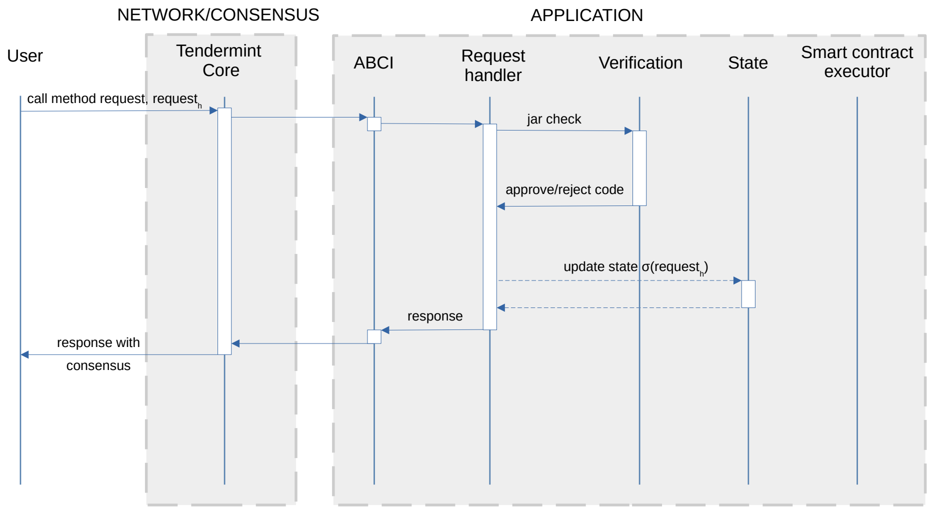

Fig. 3. Sequence diagram for code verification and installation in blockchain.

and execute smart contracts. If the code is rejected, instead, the application state is expanded with a failure response, that does not contain any code.

Fig. 4 reports an example of application state evolution. It reports the requests in full, for readability, but remember that only the hash of the requests is kept in the application state. Fig. 4 (a) shows the application state after the execution of a code installation request for which verification succeeds. The code is Java bytecode, packaged into a *jar*, *i.e.*, a zipped container of Java bytecode. The response contains the same jar (*i.e.*, the same code as the request[3]). In terms of Java, the hash of the request is the *classpath* of subsequent code executions. Fig. 4 (b) reports, instead, a request whose code fails to verify. The response does not include any code installed in blockchain. This shows that the verification rules are part of the consensus rules that determine which code installation request is valid and which must be rejected instead (Fig. 4 (a) and (b)). Hence they must be the same in every node of the network and must be deterministic.

On-chain verification performs code verification statically, only once, when the code is installed in blockchain. For instance, Fig. 4 (c) shows a subsequent request that asks to instantiate a smart contract whose code has been installed by the request in Fig. 4 (a). The request in Fig. 4 (c) uses the hash of the request in Fig. 4 (a) as its classpath and contains the parameters for calling the constructor of the smart contract. The execution of the request runs that constructor, without code verification: it has been already performed in Fig. 4 (a).

[3] The response might also contain an instrumentation of the code, as it is the case for the Java subset for programming smart contracts called Takamaka, that we use in our implementation. This is irrelevant here and we refer the interested reader to [14].

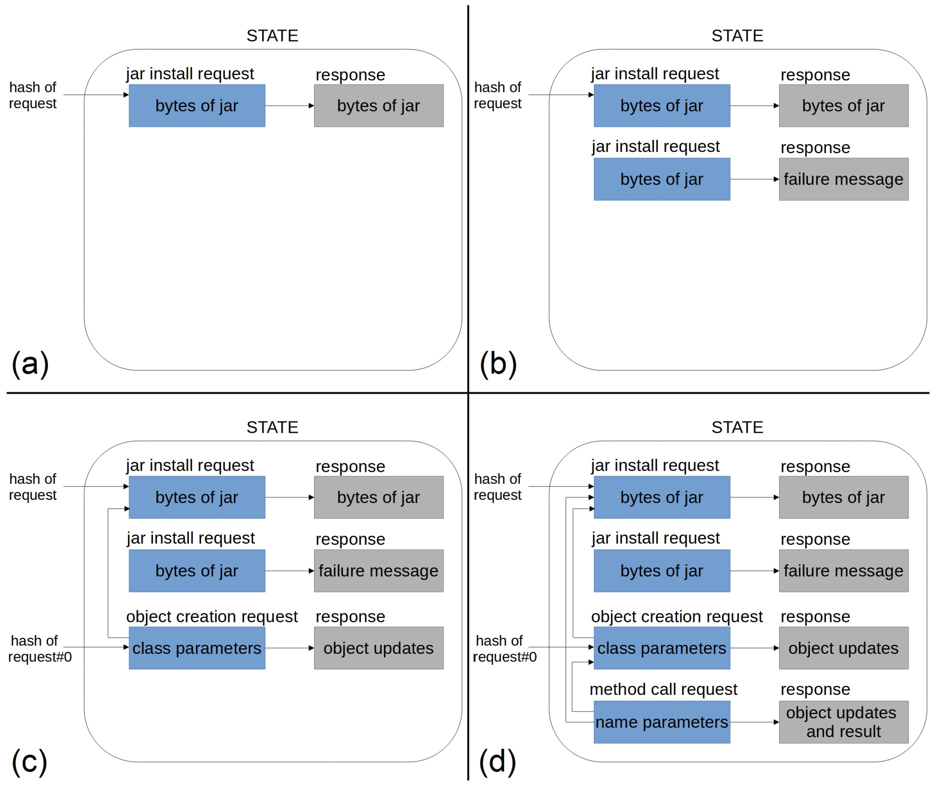

Fig. 4. The evolution of the application state during a sequence of requests.

The immutable reference *hash of request#0* is used later to refer to the new smart contract instance[4]. The state of the new smart contract is reported in the response as a set of *updates*, that is, instance fields modified during the execution of the request, including those of the smart contract instance *hash of request#0* that has been created in blockchain. Finally, Fig. 4 (d) shows the execution of a request asking to call a method on the instance of smart contract *hash of request#0*. This last request refers to both the classpath and the target instance smart contract. Its execution, in general, modifies some instance fields of objects in blockchain, that are reported as *updates* in its response. This last request does not verify the code either, since it is not a code installation request.

The rules of on-chain verification are part of the consensus rules of the blockchain, since they determine if the response of a request to install code in the blockchain is successful or failed. Hence, they determine the evolution of

[4] The index #0 refers to the first object created during the execution of a request. In general, a request can instantiate many objects, depending on the code that it executes. For simplicity, this example assumes that only one has been instantiated.

the state of the application layer and its hash, that is reported in the blocks of the underlying Tendermint blockchain, that uses it for consensus. This is the standard way of working for Tendermint. Hence, all nodes must use the same verification rules. Nodes that use different rules will be automatically excluded from the Tendermint blockchain.

3 Implementation

We have implemented on-chain verification for smart contracts written in the Takamaka subset of Java [14] (the lazy re-verification technique of Sec. 5 is still under development and we leave it for future work). The goal of Takamaka is to write smart contracts in a well-known programming language, leveraging expertise and existing mature development tools. The application layer of Takamaka is a state machine (the Tendermint *application* in Fig. 2) that executes transactions from request to response. Requests can specify the addition of a jar in the permanent state of the application, or the execution of a constructor, or of an instance or static method of code previously installed in the state. Responses include the effects of the transaction, as a set of field updates (see Fig. 4). Updates can be computed since the jar of the Java code is instrumented before being installed in blockchain, with extra code that keeps track of the affected fields of objects [14]. Determinism is ensured since only a deterministic subset of Java is allowed, restricted to a deterministic API of the Java library [15]. The state machine of Takamaka is implemented in Java and runs on a standard Java virtual machine. The state is kept in a Merkle-Patricia trie that implements a map from hash of requests to their corresponding response (Fig. 4). This trie is kept in the Xodus transactional database by JetBrains[5].

The verification module is implemented as a sequence of *checks* performed on methods and classes. Since the request of installing new code in blockchain contains the compiled bytecode only, such checks run at Java bytecode level, by using the BCEL library for Java bytecode manipulation[6]. The source code is simply not available in blockchain. Currently, Takamaka's on-chain verification performs 26 checks on every jar that gets installed in blockchain. They must all pass, or otherwise the jar will be rejected. Fig. 5 describes some of them.

We show a specific example of check now. It verifies that method `caller()` is used in the right context. That method corresponds to `msg.sender` in Solidity: it allows programmers to get a reference to the *contract* that called a method or constructor X.

[5] `https://github.com/JetBrains/xodus`
[6] `https://commons.apache.org/proper/commons-bcel`

Correct context for `@FromContract`	`@FromContract` is only applied to instance methods or to constructors of storage classes (*i.e.*, classes whose instances can be kept in blockchain).
Correct calls to `@FromContract`	`@FromContract` methods or constructors are only called from instance methods or constructors of contracts.
Correct context for `@Payable`	`@Payable` is only applied to `@FromContract` methods or constructors of contracts (since only contracts have a balance).
Correct fields in storage classes	Classes whose instances can be kept in blockchain can only have a restricted set of types for their fields.
Correct context for `caller()`	See the description in this paper.
No finalizers	Since their execution is non-deterministic in Java.
Only white-listed Java APIs	To enforce determinism (see [15]).

Fig. 5. Some of the 26 on-chain verifications currently performed by Takamaka.

The method `caller()` can be used inside the code of X only if X satisfies two constraints[7]:

1. X is annotated as `@FromContract(class)`, for some `class`;
2. the invocation of `caller()` occurs on `this`.

The rationale of constraint 1 is that `@FromContract(class)` guarantees that X can *only* be called from a contract of type `class`, or subclass, or from an external wallet whose paying account has type `class`, or subclass. Hence the caller exists. For instance, the following contract stores its creator in field `owner`. The use of `caller()` is correct here, since it occurs inside a `@FromContract` constructor:

[7] `@FromContract` and, later, `@Payable` are Java *annotations*, that is, a mechanism for adding metadata information to source and compiled code. They are irrelevant for the code executor, but can be used by code analysis and instrumentation tools.

```
import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;

public class C1 extends Contract {
  private C1 owner;

  public @FromContract(C1.class) C1() {
    owner = (C1) caller(); // ok
  }
}
```

Instead, it is incorrect to invoke `caller()` in a method or constructor not annotated as `@FromContract`, since its caller is not necessarily a contract and `caller()` would be meaningless in that case:

```
import io.takamaka.code.lang.Contract;

public class C2 extends Contract {
  public void m() {
    ... = caller(); // error at deployment time
  }
}
```

The reason of constraint 2 is that its violation lets one access the caller of other contracts, with possible logical inconsistencies and security issues. For the same reason, the use of `tx.origin` is normally an antipattern in Solidity (see *Tx.origin Authentication* in [2]). Constraint 2 holds in classes `C1` and `C2` above, but is violated below:

```
import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;

public class C3 extends Contract {
  private C3 owner;

  public @FromContract(C3.class) C3() {
    owner = (C3) caller(); // ok
  }

  public @FromContract void m() {
    ... owner.caller() ...; // error at deployment-time
  }
}
```

Fig. 6 reports our implementation of a check that verifies if a method satisfies constraints 1 and 2 above. The code has been simplified for readability: its complete version can be found in the repository of the distribution of our implementation of the runtime of Takamaka (see Sec. 4). Full understanding of the code in Fig. 6 requires knowledge about Java bytecode and BCEL, which

```
public class CallerIsUsedOnThisAndInFromContractCheck extends Check {

  public CallerIsUsedOnThisAndInFromContractCheck() {
    boolean isFromContract = annotations.isFromContract
      (className, methodName, methodArgs, methodReturnType);

    instructions()
      .filter(this::isCallToCaller)
      .forEach(ih -> {
        if (!isFromContract)
          issue(new CallerOutsideFromContractError(inferSourceFile(), methodName, lineOf(ih)));

        if (!previousIsLoad0(ih))
          issue(new CallerNotOnThisError(inferSourceFile(), methodName, lineOf(ih)));
      });
  }

  private boolean previousIsLoad0(InstructionHandle ih) {
   Instruction ins = ih.getPrev().getInstruction();
   return ins instanceof LoadInstruction && ((LoadInstruction) ins).getIndex() == 0;
  }

  private final static String TAKAMAKA_CALLER_SIG = "()Lio/takamaka/code/lang/Contract;";

  private boolean isCallToCaller(InstructionHandle ih) {
    Instruction ins = ih.getInstruction();
    if (ins instanceof InvokeInstruction) {
      InvokeInstruction invoke = (InvokeInstruction) ins;
      ReferenceType receiver;

      return "caller".equals(invoke.getMethodName())
        && TAKAMAKA_CALLER_SIG.equals(invoke.getSignature())
        && (receiver = invoke.getReferenceType()) instanceof ObjectType
        && classLoader.isStorage(((ObjectType) receiver).getClassName());
    }
    else
      return false;
  }
}
```

Fig. 6. The on-chain check for a correct use of `caller()`.

is outside the scope of this paper. Nevertheless, it is possible to understand the structure of the code: the constructor of the check scans the stream of Java bytecode instructions of the method (`instructions()`), filters those that call a method named `caller` that returns a contract, and checks two conditions for each of them (with the two `if`'s inside the `forEach`): the method must be annotated as `FromContract` (constraint 1 above) and the invocation must be immediately preceded by an `aload_0` bytecode instruction. The latter is Java bytecode for pushing `this` on the stack, as receiver of the call to `caller()` (constraint 2 above). If any of the `if`'s is satisfied, an issue is generated, which will later reject the installation of the code in blockchain.

4 Experiments

We have implemented our on-chain verification for the Takamaka subset of Java, inside its runtime that works as a Tendermint application. It is an actual blockchain running on Tendermint, that can be programmed with smart contracts written in Java. Our implementation is part of a larger project, called Hotmoka, whose long-term goal is to use the Takamaka language for programming both blockchains and IoT devices. We have created three scripts that request to install in blockchain the examples from Sec. 3. We have also created a test that installs a smart contract and uses it to run many transactions, to check the scalability of the technique and evaluate the difference when on-chain verification is on or off. Readers who want to run the experiments and inspect the results can download the code[8] and follow the instructions in the `WTSC21.txt`. That repository contains also the code of the 26 checks of on-chain verification (including that in Fig. 6).

The first experiment starts a blockchain of a single node and runs a script that connects to the node and installs a jar containing class `C1` from Sec. 3. The result is successful:

```
Connecting to the blockchain node at localhost:8080... done
Installing the Takamaka runtime in the node... done
Installing C1 in the node... done (on-chain verification succeeded)
C1.jar installed at address ee848b5bc7fd8283ab01b5977970e71f548...
```

The subsequent experiment installs `C2` instead. The attempt to install the code in blockchain will fail since on-chain verification fails:

```
Connecting to the blockchain node at localhost:8080... done
Installing the Takamaka runtime in the node... done
Installing C2 in the node...
Exception in thread "main" io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException: C2.java:8
caller() can only be used inside a @FromContract method or constructor
```

The third experiment performs the same operation with class `C3`. This attempt will fail since on-chain verification fails:

```
Connecting to the blockchain node at localhost:8080... done
Installing the Takamaka runtime in the node... done
Installing C3 in the node...
Exception in thread "main" io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException: C3.java:14
caller() can only be called on "this"
```

In order to evaluate the scalability of our technique, we have created a smart contract that creates and funds a pool of 500 externally-owned accounts and allows one to determine which is the *richest* among them (has highest balance). We have written a JUnit test that installs that smart contract in blockchain and uses it to create and fund the 500 accounts, execute $1,000$ random money transfers between them and ask for the richest. This process is repeated ten times. The execution time of this test is 158.19 seconds on our machine (Intel

[8] `git clone -branch wtsc21 https://github.com/Hotmoka/hotmoka.git`

Core i3-4150, 16GB of RAM, running Ubuntu Linux 20.04.1). In total (including code installation and account creation) the test runs $10,020$ transactions, that is, it performs 63.34 transactions per second. By turning on-chain verification off, the same test runs in 156.95 seconds, that is, it performs 63.84 transactions per second. These numbers have been computed as an average over five executions of the test. This shows that on-chain verification increases the execution time of the test by only 0.79%.

5 Evolution of Code Verification

This section shows that a change in the verification rules requires to re-verify all code installed in blockchain and that this can be performed lazily, on-demand.

Sec. 2 stated that code verification is only performed when code is installed in blockchain. However, that is true only under the unrealistic assumption that the verification module never changes. In practice, that module will be updated eventually, to include new verification rules or to improve the precision of already existing rules. When a new version is deployed, it becomes necessary to update all nodes to that version (or at least all validators), or otherwise consensus might be lost. A change in the verification rules, if deployed on a subset of the network only, entails that the updated nodes might accept a request that the non-updated nodes might reject instead, or vice versa.

All approaches to a network update can be used here. The novelty, however, is that some code that was successfully verified with the previous version of the verification module might be rejected with its current version, or vice versa. Hence, there must be a mechanism that enforces that the execution of some code in blockchain occurs only if that code passes the *current* verification rules. Conceptually, this means that an update of the verification module triggers a re-verification of all code previously successfully installed in blockchain. In practice, this cannot be performed, since it would be extremely expensive and would hang the nodes for a long time. Our solution, that we are going to describe, is to lazily re-verify the code on-demand, when it is asked to run. This amortizes the cost of re-verification. Moreover, [10] shows that only 0.05% of all contracts installed in Ethereum are involved in 80% of the transactions. Hence, a lazy approach avoids the re-verification of code that might actually never run again.

In order to implement this lazy re-verification approach, we expand the information in the *response* of a successful code installation *request* (Fig. 4 (a)). Namely, together with the installed code, *response* is enriched with a numerical tag $\tau(\mathit{response})$, *i.e.*, the version of the verification module that has been used to verify the code inside *response*. The sequence diagram in Fig. 7 shows the workflow for lazy code re-verification. Assume that a request arrives, that requires to run code referred with the hash $\mathit{request}_h$ of a previous, successful code installation *request* (as in Fig. 4 (c) and (d)). The node finds out that $\sigma(\mathit{request}_h) = \mathit{response}$ has a verification tag $\tau(\mathit{response})$ and compares it with the current version τ of the verification module. There are two possibilities:

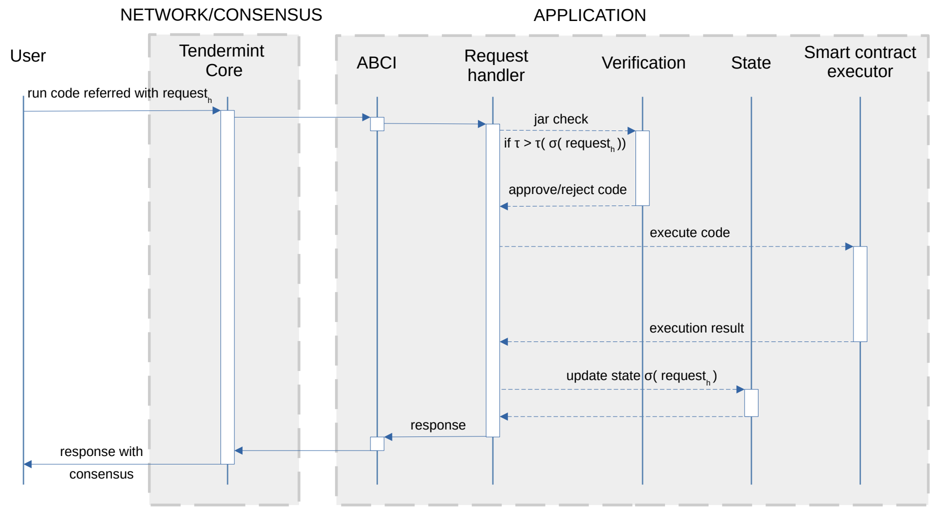

Fig. 7. Sequence diagram for lazy code re-verification.

1. $\tau = \tau(response)$: the code was verified with the current version of the verification module, it does not need re-verification and can be run immediately;
2. $\tau > \tau(response)$: the code was verified with an old version of the verification module; it must be re-verified before being run.

In the second case, the node verifies the code again, using the current version τ of the verification module. This is possible since *response* includes that code (Fig. 4 (a)). A new response $response'$ will be computed (successful, having τ as verification module version, or failed) and the application state is updated as $\sigma(request_h) = response'$. The use of $request_h$ in future requests will not re-verify the code, until a newer version of the verification module is installed. The update is possible since it occurs in the state, not in the blockchain, whose blocks are immutable.

It is important to note that $response'$ might state that reverification failed, because the old code passed the previous verification rules but not the new ones. In that case, the execution of the code will fail, since its classpath is not valid anymore. This means that a smart contract might work today, but might stop working tomorrow, if updated verification rules reject its code. In theory, the converse is also possible: the same contract might be reactivated after tomorrow, if another change in the verification rules replaces a failed response with a successful response. However, we have decided to forbid this second scenario, since it might be surprising for users.

6 Discussion

To our knowledge, this paper defines and implements the first on-chain code verification for smart contracts, that allows the same blockchain to reject the code that does not pass a set of verification rules. From this point of view, the technique is related to continuous integration, that builds and deploys code only if it passes all compilation and testing requirements. The main difference is that smart contracts cannot be replaced or debugged once installed in blockchain.

Some blockchains, such as Ethereum, apply a notion of *transparency* [10], that lets one store in blockchain the source code of the smart contracts, to guarantee that it actually compiles into their bytecode. But this is only an optional technique that ensures that bytecode and source code match: no code verification is applied.

The specific technique for updating the consensus rules of a network, after a change in the verification rules (Sec. 5), is orthogonal to our work. In Cosmos, the government module supports such an update, with (dis-)incentives to minimize misconduct within the participants. *Polkadot* delegates updates to periodic referendums among stakeholders[9]. Algorand [5] triggers an update if a large majority of block proposers declare to be ready for that.

On-chain verification must be efficient, in order not to block the nodes of the network. Our experiments (Sec. 4) show that the time of analysis is largely dominated by the time of block creation, also because smart contracts are typically small. Nevertheless, the on-chain application of powerful static analyses, such as those currently running, for instance, on Java desktop applications [13], seems challenging. On-chain verification must be understood as a mandatory, defensive verification technique, rather than as a replacement for off-chain verification.

In Sec. 5, a change of the verification rules triggers the re-verification of code already in blockchain. This might not be the best choice, since it might disable some smart contracts already in blockchain and lock their funds. Moreover, a change of the verification rules might be opposed by a large number of users, if it affects some highly popular contract. Future work will investigate linguistic primitives and programming patterns that allow funds to be unlocked or specify that some contract should *not* be re-verified after a verification rules change.

References

1. A. M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies.* O'Reilly, 2nd edition, 2017.
2. A. M. Antonopoulos and G. Wood. *Mastering Ethereum: Building Smart Contracts and Dapps.* O'Reilly, 2018.
3. N. Atzei, M. Bartoletti, and T. Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *6th International Conference on Principles of Security and Trust (POST'17)*, volume 10204 of *Lecture Notes in Computer Science*, pages 164–186, Uppsala, Sweden, April 2017. Springer.

[9] See `https://wiki.polkadot.network/docs/en/learn-governance`

4. V. Buterin. Ethereum Whitepaper, 2013. Available at `https://ethereum.org/en/whitepaper/`.
5. J. Chen and S. Micali. Algorand: A Secure and Efficient Distributed Ledger. *Theoretical Computer Science*, 777:155–183, 2019.
6. J. Feist, G. Grieco, and A. Groce. Slither: A Static Analysis Framework for Smart Contracts. In *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB@ICSE'19)*, pages 8–15, Montreal, QC, Canada, May 2019. IEEE / ACM.
7. G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*, pages 557–560, USA, July 2020. ACM.
8. J. Kwon. Tendermint: Consensus without Mining. Available at `https://tendermint.com/static/docs/tendermint.pdf`, 2014.
9. S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Available at `https://bitcoin.org/bitcoin.pdf`, 2008.
10. G. A. Oliva, A. E. Hassan, and Z. M. Jiang. An Exploratory Study of Smart Contracts in the Ethereum Blockchain Platform. *Empirical Software Engineering*, 25(3):1864–1904, 2020.
11. N. Popper. A Hacking of More than $50 Million Dashes Hopes in the World of Virtual Currency. The New York Times, 2016-06-18.
12. J. Sedlmeir, H. U. Buhl, G. Fridgen, and R. Keller. The Energy Consumption of Blockchain Technology: Beyond Myth. *Business & Information Systems Engineering*, 62(6):599–608, 2020.
13. F. Spoto. The Julia Static Analyzer for Java. In *23rd Static Analysis Symposium (SAS'16)*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57, Edinburgh, UK, September 2016. Springer.
14. F. Spoto. A Java Framework for Smart Contracts. In *3rd Wokshop on Trusted Smart Contracts (WTSC'19)*, volume 11599 of *Lecture Notes in Computer Science*, pages 122–137, St. Kitts and Nevis, February 2019. Springer.
15. F. Spoto. Enforcing Determinism of Java Smart Contracts. In *4th Wokshop on Trusted Smart Contracts (WTSC'20)*, volume 12063 of *Lecture Notes in Computer Science*, pages 568–583, Kota Kinabalu, Malaysia, February 2020. Springer.