Cluster Computing
https://doi.org/10.1007/s10586-022-03688-y

On the use of generic types for smart contracts

Fausto Spoto[1] · Sara Migliorini[1] · Mauro Gambini[1] · Andrea Benini[1]

Received: 7 March 2022 / Revised: 3 May 2022 / Accepted: 4 July 2022

Abstract
This paper shows that generic types (*generics*) are useful for writing more abstract and more general smart contracts, but this comes with some security risks, reporting a concrete security issue found while using generics for writing smart contracts that implement *shared entities* for the Hotmoka blockchain. That issue can be used to steal the remuneration of validator nodes. This paper proposes a patch based on appropriate code rewriting. Namely, smart contracts are pieces of code that are deployed and executed in the context of a blockchain infrastructure in order to automatically enforce some effects when particular events occur. The writing of smart contracts is a complex and critical activity that can benefit from the use of high-level features of programming languages, and generics is one of them. In many programming languages, such as Java, generics are implemented by *erasure*, i.e. replaced by their upper bound type during compilation into bytecode. This is safe at source level, since the compiler takes care of checking that types are correct, before erasure. However, the erased types of the generated bytecode are consequently weaker. In a permissionless blockchain, where every user can call the bytecode of smart contracts installed by other users, these weaker types pose a risk of attack.

Keywords Smart contracts · Generics · Erasure

1 Introduction

In recent years, blockchain technology has rapidly emerged as a powerful tool for supporting the development of many and innovative services and infrastructures. Blockchain-enabled applications are spreading across diverse sectors such as supply chain, business, healthcare, IoT, privacy, and data management [1]. A blockchain is essentially a *distributed ledger*, namely a database replicated across different locations and synchronized by multiple independent participants. Blockchains exploit the redundant, concurrent execution of the same transactions on a decentralized network of many machines, in order to enforce their execution in accordance with a set of predefined rules. Namely, blockchains make it hard, for a single machine, to disrupt the semantics of transactions or their ordering: a misbehaving single machine gets immediately put out of consensus and isolated.

That is, the key innovation introduced by this technology is a mechanism able to reach an emergent agreement about a global state without the need for a central authority. Morever, another peculiarity is that the consensus is not explicit, because there is not a fixed moment when it occurs.

The rules of blockchain transactions are specified by *smart contracts*, that are code written in a variety of programming languages. To the best of our knowledge, none of them allows generic types (*generics*) and, in any case, nothing has been published about the opportunity, but also the risks of using generics for writing smart contracts. The contribution of this paper is exactly to show a real-life use of generics for an actual smart contract contained in the support library of the Takamaka language [2, 3], and to demonstrate that a naïve use of Java generics can lead to a code security vulnerability that allows an attacker to earn money by exploiting someone else's work, with both

Fausto Spoto, Sara Migliorini, Mauro Gambini and Andrea Benini have contributed equally to this work.

✉ Sara Migliorini
sara.migliorini@univr.it

Fausto Spoto
fausto.spoto@univr.it

Mauro Gambini
mauro.gambini@univr.it

Andrea Benini
andrea.benini@studenti.univr.it

[1] Department of Computer Science, University of Verona, Verona, Italy

Published online: 09 August 2022

economical and legal side effects. This paper provides a fix to that specific issue, by proposing a re-engineering of the code that forces the compiler to generate defensive checks. More generally, this paper can be useful for the definition of future bytecode languages for smart contract languages, by learning from the weaknesses of Java bytecode, in particular from those related to the compilation of generics.

Historically, programming languages for specifying blockchain transactions started with Bitcoin [4, 5], the first blockchain's success story. Here transactions are programmed in a non-Turing complete bytecode language, with no notion of generic types, almost exclusively used to implement transfers of units of coins between *accounts*, providing a totally decentralized P2P digital cash system based on a distributed public ledger. A few years after Bitcoin, another blockchain, called Ethereum [6, 7], introduced the possibility of programming transactions in an actual, imperative and Turing-complete programming language, called Solidity, also missing generic types. The major innovation of Ethereum is the construction through its nodes of a distributed *world computer* that can run general-purpose code. Indeed, if the term *distributed ledger* is usually used to describe blockchains like Bitcoin, Ethereum is often defined as a *distributed state machine*. Solidity's code is organized in *smart contracts*, namely pieces of code that are stored in the blockchain and are executed when a particular event occurs, e.g. when a transaction is scheduled. From a theoretical point of view, a smart contract is essentially an agreement between two or more parties that can be automatically enforced without the need for a trustworthy intermediary [8]. Through smart contracts, Ethereum's transactions can hence execute much more than coin transfers. In this case the global shared state is given by a set of objects that are persisted and manipulated in the same way by all nodes in the blockchain through the execution of the same object constructors and methods.

The world computer built by Ethereum is known as Ethereum Virtual Machine (EVM) and is the platform where accounts and smart contracts live and are executed. Solidity is a high-level programming language and smart contracts need to be compiled into bytecode to be executed inside the EVM. At this regard, observe that, in Solidity's bytecode, non-primitive values are referenced through a very general `address` type. For instance, a Solidity method `child(Person p, uint256 n) returns Person` actually compiles into `child(address p, uint256 n) returns address`, losing most type information [9]. Since, at run time, it is the bytecode that gets executed, everything can be passed for `p`, not just a `Person` instance, as illustrated in Fig. 1. The compiler cannot even enforce strong typing by generating defensive type instance checks and casts, because values are unboxed in Ethereum: they have no attached type information at run time, they are just numerical *addresses*. It follows that, inside the `child` method, an eventual call to a `Person`'s method on `p` might actually execute any arbitrary code, if `p` is not a `Person`. In other words, Solidity is not strongly typed. Consequently, it is highly discouraged, in Solidity, to call methods on parameters passed to another method, such as on `p` passed to `child`, since an attacker can pass crafted objects for `p`, with arbitrary implementations for their methods, which can result in the unexpected execution of dangerous code. This actually happened in the case of the infamous DAO hack [10], that costed millions of dollars.

Strong typing is one of the reasons that pushed towards the adoption of *traditional* programming languages for smart contracts. For instance, the Cosmos blockchain [11] uses Go. The Hotmoka blockchain [12] uses a subset of Java for smart contracts, called Takamaka [2, 3]. Hyperledger [13] allows Go and Java. Another reason is the availability of modern language features, that are missing in Solidity, such as generics. They are a powerful and very useful facility for programming smart contracts, since they allow one to personalize the behavior of such contracts and partially overcome their inherent incompleteness [8]. Through the use of generics, it is possible to provide to users a set of predefined contract templates that they can extend and specialize with lower programming skills, but higher knowledge about the specific application domain. Generics are based on the use of type placeholders in order to produce parametrized code, that can be instantiated for each concrete type provided for the placeholders. However, strong typing and generics are two intertwined language features that have to be carefully considered when smart contracts are implemented and their bytecode is subsequently deployed on a blockchain. For instance, in Java source code, generics are strongly typed, if no *unchecked operations* are used [14], as it will always be the case in this paper. However, generics might have security issues at the level of compiled Java code and this paper originated from a real issue that has been found in our code.

The remainder of this paper is organized as follows. Section 2 discusses the management of generics in Java. Section 3 presents the basic notions about the Takamaka language for smart contracts in Java. Section 4 shows our real-life Java smart contracts for shared entities, that use generics. Section 5 shows the instantiation of the shared entities to implement the validators' set of a proof of stake blockchain. Section 6 shows that a naïve deployment of a subclass of the validators' set leads to a code vulnerability due to the way generics are compiled. Section 7 presents a fix to that vulnerability. Section 8 discusses some related work. Section 9 concludes.

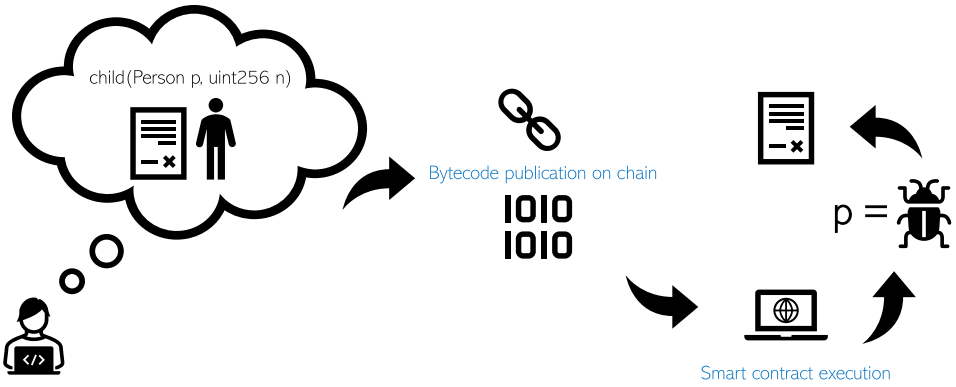

Fig. 1 Example of possible problem that can occur in Solidity due to the absence of a strong typing mechanism

This paper is a revised and extended version of [15]. In comparison to that paper, Sects. 3 and 5 are new; while all other sections have been expanded with several additional details and enriched with many explanatory figures.

2 Generics implementation in programming languages

There exist two common ways to implement generics in a programming language, that are often described in literature as *heterogeneous* and *homogeneous* [16]. In the heterogeneous approach, the code is duplicated and specialized for each instance of the generic parameters; this is the approach adopted by C++ *templates*. Conversely, the homogeneous approach is that provided by Java and .Net; in this case, only one instance of the code is maintained and shared by all generic instances. This implementation is based on the type *erasure* mechanism, where the generic parameter is replaced by the upwards bound of each instance, mostly often `Object`. Even though the heterogeneous approach is the safest, it is rarely applied, in particular in resource-constrained applications, because the code size may dramatically increase as a consequence of duplication [17]. For code in blockchain, the heterogeneous approach obliges one to reinstall all instantiations of the generic code, with extra costs of gas, which makes it impractical. Conversely, the homogeneous approach ensures a smaller consumption of resources.

In order to understand the mechanism of erasure, consider for instance the interface `SharedEntity` in Fig. 7 and its method `accept`. The functionality of `SharedEntity` will be discussed later (Sect. 4). Here, it is relevant to consider only how its generic type parameters get compiled. Namely, `SharedEntity` uses two generic type parameters `S` and `O`, that must be provided whenever a client creates a concrete implementation of the interface. Such generic parameters have an upper bound: `S` can only be a subtype of `PayableContract`, while `O` can only be a subtype of `Offer<S>`. If one checks the bytecode generated for `SharedEntity`, she will see that `accept` is declared, in bytecode, as `void accept(BigInteger amount, PayableContract buyer, Offer offer)`, that is, the two type variables `S` and `O` have been *erased* and replaced with their respective upper bound, as illustrated in Fig. 2.

Erasure weakens the type information of the compiled code. It is the responsibility of the compiler to guarantee that types are still respected, in all implementations of `SharedEntity`. In Java, the compiler guarantees type correctness and the Java language remains strongly-typed, also in the presence of generic types, if no *unchecked operations* are performed [14] (such as casts to generic types, that are unchecked for a limitation of the Java bytecode). However, this guarantee applies to Java source code compiled by the Java compiler, not to bytecode that can be generated manually, in order to attack instances of the `SharedEntity` class, as shown later.

3 The Takamaka language for smart contracts

This section gives a short introduction to the Takamaka subset of Java that this paper uses for writing smart contracts. This language has been introduced in [2]. A full tutorial is available online, as part of the documentation of the Hotmoka blockchain that runs smart contracts written in Takamaka [18]. This section introduces only the essential notions that are needed to understand the subsequent sections. The hierarchy of the classes described in this section is in Fig. 3. In the following, a simplified presentation of the code of some of such classes will be reported. The full code is in the Github repository of Hotmoka [18].

Takamaka implements objects persisted in blockchain as subclasses of the class `io.takamaka.code.lang.Storage`. This is the main difference with other attempts at using Java for writing smart contracts: the programmer does not code the serialization and

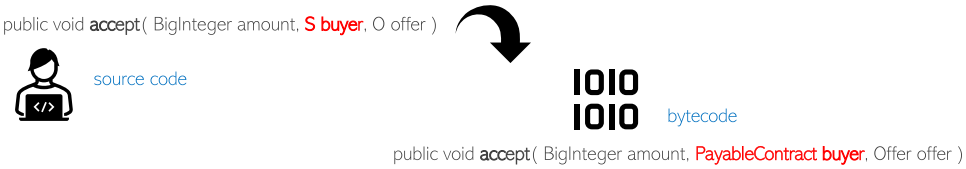

Fig. 2 Example of generics implementation by erasure in Java

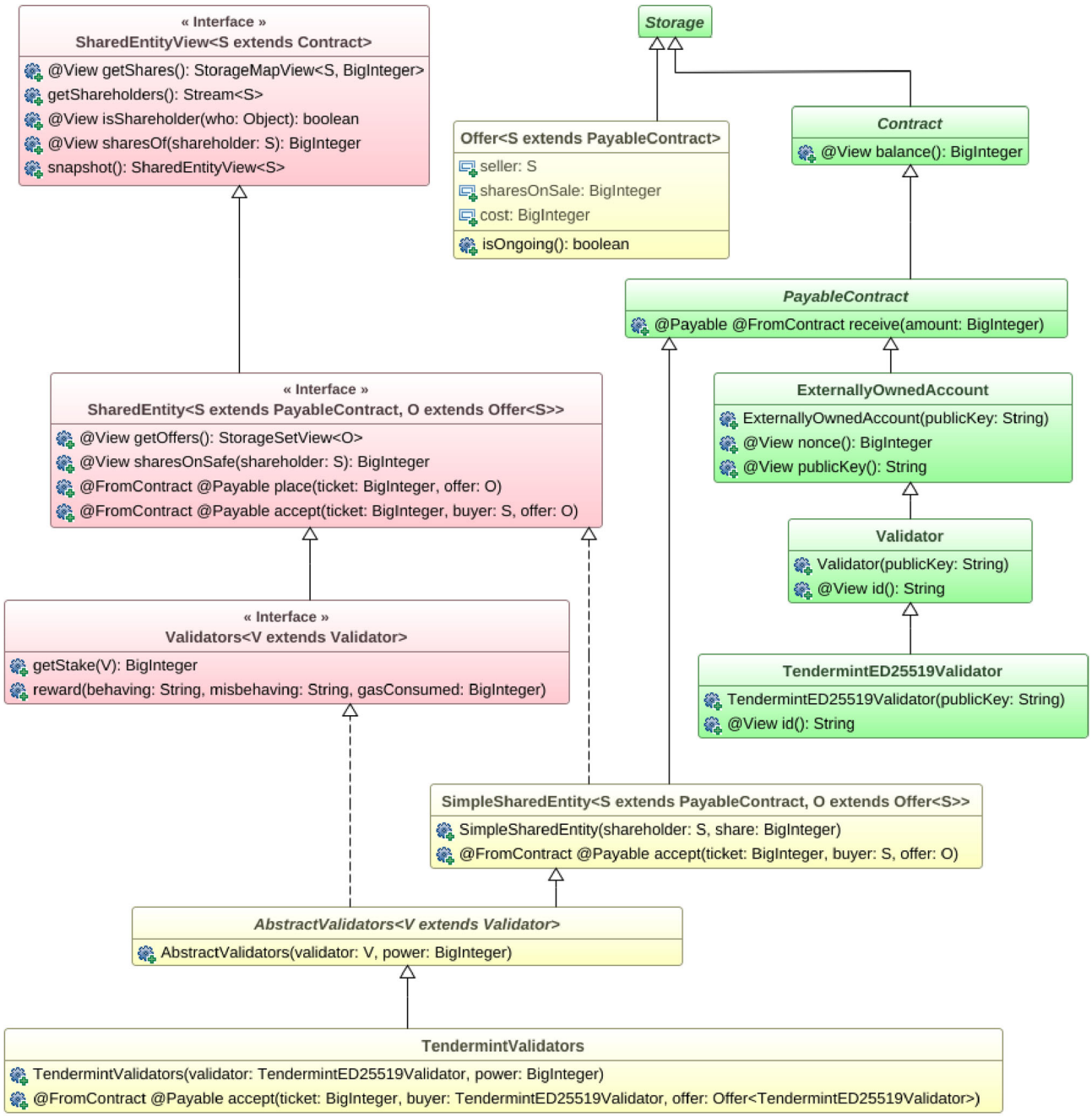

Fig. 3 The hierarchy of Takamaka classes that implement accounts, shared entities and validators. Their source code can be found inside the Java project https://github.com/Hotmoka/hotmoka/tree/master/io-takamaka-code

deserialization of objects into a *keeper* or a key/value map, but simply extends `Storage` and objects get persisted automatically *out of magic*. In this sense, Takamaka follows the approach of Solidity, but using Java.

The `io.takamaka.code.lang.Contract` class implements objects that can be persisted in blockchain *and* have a balance. Therefore, they can receive and provide payments. Their balance is available through a `balance` method. This is a `@View` method, meaning that it can be called without paying gas (the measure of execution cost), since such methods cannot have side-effects and consequently do not modify the storage of the blockchain. Payments can be received only through methods annotated as `@Payable`. The `Contract` superclass has no such methods, but subclasses may have. For instance, its `io.takamaka.code.lang.PayableContract` subclass has a method `receive` to receive payments from its caller. Many methods (inclusing all `@Payable` methods) need to identify their caller. This is done by adding the `@FromContract` annotation, that guarantees that the caller is a contract, available inside the method as `caller()`.

Method calls started from outside the blockchain (for instance, from a client such as a wallet or from a web application), must specify an already existing `ExternallyOwnedAccount` as caller. This account will pay for the gas of the execution. The blockchain will accept the call only if is signed with the private key that matches the public key provided to the constructor of the account when it was created. Method `publicKey` allows one to recover that public key and method `nonce` allows one to get a progressive identifier that can be used to distinguish successive calls with the same account, to force their order of execution and to avoid replaying. All that is very similar to Solidity, except for the fact that externally owned accounts are actual Java objects inside the blockchain, not just an abstraction of a public key. An exemplification of a call made to a `PayableContract` is reported in Fig. 4

Neither the Takamaka language nor the Hotmoka blockchain dictate a specific consensus mechanism. Both proof of work and proof of stake can be used, for instance. In particular, if proof of stake is used, then each validator node of the blockchain must specify a `io.takamaka.code.governance.Validator` object, that plays the role of the banking account where the validation rewards of the node get accumulated (see Fig. 5). It is a special externally owned account, with an extra `id` method that provides the identifier of the validator node inside the blockchain network. This identifier depends from the specific network. For instance, the subclass `io.takamaka.code.governance.tendermint.TendermintED25519Validator` implements `id` as for the Tendermint blockchain engine [19], that is, as the first 40 hexadecimal digits of the sha256 digest of the Base64-encoded public key (see its code in Fig. 5).

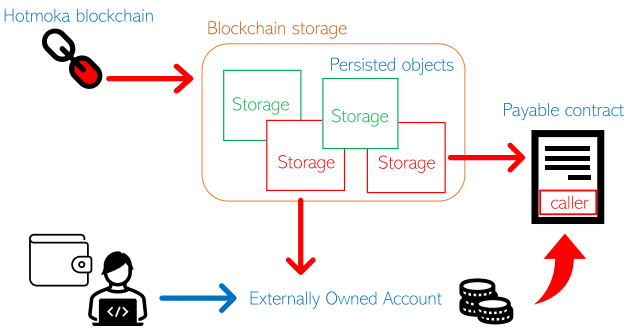

Fig. 4 Exemplification of the Takamaka persistent objects stored in the blockchain

4 A generic shared entities implementation

A *shared entity* is a concept that often arises in blockchain applications. Namely, a shared entity is something divided into *shares*. Participants, that hold shares, are called *shareholders* and can dynamically sell and buy shares. An example of a shared entity is a corporation, where shares represent units of possess of the company. Another example is a voting community, where shares represent the voting power of each given voter. A further example is the set of the validator nodes of a proof of stake blockchain, where shares represent their voting power and remuneration percentage.

In general, two concepts are specific to each implementation of shared entities: who are the potential shareholders and how offers for selling shares work. Therefore, one can parameterize the interface of a shared entity with two type variables: `S` is the type of the shareholders and `O` is the type of the sale offers of shares.

The `SharedEntityView` interface at the top of the hierarchy in Fig. 3 defines the read-only operations on a shared entity. This view is *static*, in the sense that it does not specify the operations for transfers of shares. Therefore, its only type parameter is `S`: any contract can play the role of the type for the shareholders of the entity. Method `getShares` yields a snapshot of the current shares of the entity (who owns how much). Method `getShareholders` yields the shareholders. It is not `@View`, since it creates a new stream, which is a side-effect. Method `isShareholder` checks if an object is a shareholder. Method `sharesOf` yields the number of shares of a shareholder. As typical in Takamaka, a `snapshot` method allows one to create a frozen read-only copy of an entity (in constant time), useful when an entity must be queried from a client without the risk of race conditions if another client is modifying the same entity concurrently.

The `SharedEntity` subinterface adds methods for transfer of shares (see Fig. 7). It includes an inner class `Offer` that models sale offers: it specifies who is the seller

```
public class Validator extends
    ExternallyOwnedAccount {

  public Validator(String publicKey) {
    super(publicKey);
  }

  public @View String id() {
    return publicKey(); // subclasses may
        redefine
  }
}

public final class TendermintED25519Validator
      extends Validator {
  private final String id;

  public TendermintED25519Validator
    (String publicKey) {
    super(publicKey);

    MessageDigest sha256 =
      MessageDigest.getInstance("SHA-256");
    sha256.update(Base64.getDecoder().
      decode(publicKey()));
    this.id = bytesToHex(sha256.digest()).
      substring(0, 40);
  }

  @Override @View public final String id() {
    return id;
  }
}
```

Fig. 5 The account of a validator and its specialization for a Hotmoka blockchain based on Tendermint

of the shares, how many shares are being sold, the requested price and the expiration of the offer. Method `isOngoing` checks if an offer has not expired yet. Implementations can subclass `Offer` if they need more specific offers. Offers can be placed on sale by calling the `place` method with a sale `offer` (see Fig. 6). This method is annotated as `@FromContract` since the caller must be identified (or otherwise anybody could sell the shares of anybody else) and as `@Payable` so that implementations can require to pay a `ticket` to place shares on sale. The sale offer is passed as a parameter to `place`, hence it must have been created before calling that method. The set of all sale offers is available through `getOffers`. Method `sharesOnSale` yields the cumulative number of shares on sale for a given shareholder. Who wants to buy shares calls method `accept` with the accepted `offer` and with itself as `buyer` (the reason will be explained soon) and becomes a new shareholder or increases its cumulative number of shares (if it was a shareholder already). Also this method is `@Payable`, since its caller must pay `ticket` $\geq$ `offer.cost` coins to the seller. This means that shareholders must be able to receive payments and that is why `S extends PayableContract`: only `PayableContracts` are guaranteed to have a `receive` method in Takamaka.

As said before, the annotation `@FromContract` on both `place` and `accept` enforces that only contracts can call these methods. These callers must be (old or new) shareholders, hence they must have type `S`. Therefore, one would like to write `@FromContract(S.class)`. Unfortunately, Java does not allow a generic type variable `S` in the syntax `S.class`. Due to this syntactical limitation of Java, the best that can be written in Fig. 7 is `@FromContract(PayableContract.class)`, which allows *any* `PayableContract` to call these methods, not just those of type `S`. Since the syntax of the language does not support the needed abstraction, one has to program explicit dynamic checks in code, as shown later, and this will be the reason of the parameter `buyer` in `accept`.

Figure 8 shows a portion of the code of our `SimpleSharedEntity` implementation of the `SharedEntity` interface in Fig. 7, that uses two fields: `shares` maps each shareholder to the amount of shares the it holds and `offers` collects the offers that have been placed. The constructor initially populates the map `shares` with the initial shareholder. Other shareholders can be added later, by buying shares.[1] Method `sharesOf` simply accesses `shares`, by using zero as default. Method `place` requires its `caller()` to be the seller identified in the `offer`. This forbids shareholders to sell shares on behalf of others. Moreover, this guarantees that the caller has type `S`, the type of `offer.seller`. As it has been said before, this cannot be expressed with the syntax of the language. Method `place` further requires the seller to be a shareholder with at least `offer.sharesOnSale` shares not yet placed on sale. This forbids to oversell more shares than one owns. At the end, `place` adds the `offer` to the set of `offers`. Method `accept` requires that who calls the method must be `buyer`. Hence, successful calls to `accept` can only pass the same caller for `buyer`. This is a trick to enforce the caller to have type `S`, since the syntax of the language does not allow one to express it, as explained before. Then `accept` requires the `offer` to exist, to be still ongoing and to cost no more than the `amount` of money provided to `accept`. If that is the case, the `offer` is removed from the `offers`, shares are moved from seller to buyer (code not shown in Fig. 8) and the seller of the `offer` receives the required price `offer.cost`.

[1] In the real code, the class has constructors to create shared entities with a *set* of initial shareholders. This paper reports a simplification of the actual code.

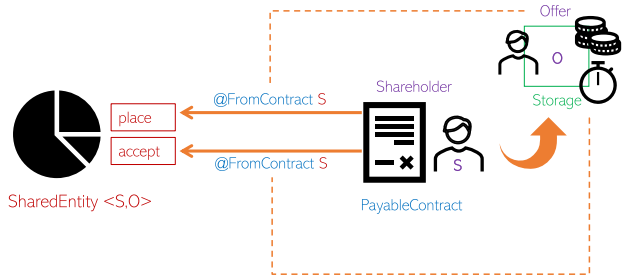

Fig. 6 Exemplification of a Takamaka shared entity and of its connections with the objects persisted in the blockchain

5 Blockchain validators set as a shared entity

The Hotmoka blockchain is built over Tendermint [19], a generic engine for replicating an application over a network of nodes. In our case, the application is an executor of smart contracts in Java, such as that in Fig. 8. Tendermint is based on a proof of stake consensus, which means that a selected dynamic subset of the nodes is in charge of validating the transactions and voting their acceptance. As already said, Hotmoka models validator nodes as `Validator` objects, that are externally owned accounts with an extra identifier. In the specific case of a Hotmoka blockchain built over Tendermint, validators are `TendermintED25519Validator` objects whose identifier is derived from their ed25519 public key (see Fig. 5). This identifier is public information, reported in the blocks or easily eavesdropped. Tendermint applications can implement their own policy for rewarding or changing the validators' set dynamically.

The set of the validator nodes of a blockchain network is an example of a shared entity. Namely, each such validator owns an amount of validation power, that corresponds to the shares of a shareholder. Validation power can be sold and bought, exactly as shares. Consequently, the `Validators` interface in Fig. 3 (reported in Fig. 9) extends the `SharedEntity` interface, fixes the shareholders to be instances of `Validator` and adds two methods: `getStake` yields the money at stake for each given validator (if the validator misbehaves, its stake will be reduced or *slashed*); and `reward`, that is called by the blockchain itself at the end of each block creation: it distributes the cost of the gas consumed by the transactions of the block, to the well-behaving validators, and slashes the stakes of the misbehaving validators.

The `AbstractValidators` class implements the validators' set and the distribution of the reward and is a subclass of `SimpleSharedEntity` (see Figs. 9, 10). Shares are voting power in this case. Its subclass `TendermintValidators` restricts the type of the validators to be `TendermintED25519Validator`. At each block committed, Hotmoka calls the `reward` method of `Validators` in order to reward the validators that behaved correctly and slash those that misbehaved, possibly removing them from the validators' set. They are specified by two strings that contain the identifiers of the validators, as provided by the underlying Tendermint engine.

Fig. 7 A simplified part of our shared entity interface

```
public interface SharedEntity<S extends PayableContract, O extends Offer<S>>
  extends SharedEntityView<S> {

  @View StorageSetView<O> getOffers();

  @View BigInteger sharesOnSaleOf(S shareholder);

  @FromContract(PayableContract.class) @Payable
  void place(BigInteger ticket, O offer);

  @FromContract(PayableContract.class) @Payable
  void accept(BigInteger ticket, S buyer, O offer);

  class Offer<S extends PayableContract> extends Storage {
    public final S seller;
    public final BigInteger sharesOnSale;
    public final BigInteger cost;
    public final long expiration;

    public Offer(S seller, BigInteger sharesOnSale, BigInteger cost,long duration) {
      this.seller = seller;
      this.sharesOnSale = sharesOnSale;
      this.cost = cost;
      this.expiration = now() + duration;
    }

    @View public boolean isOngoing() {
      return now() <= expiration;
    }
  }
}
```

Fig. 8 A simplified part of our implementation of the shared entity interface

```
public class SimpleSharedEntity
    <S extends PayableContract,O extends Offer<S>>
    extends PayableContract implements SharedEntity<S,O> {

  private StorageTreeMap<S,BigInteger> shares = new StorageTreeMap<>();
  private StorageSet<O> offers = new StorageTreeSet<>();

  public SimpleSharedEntity(S shareholder, BigInteger share) {
    addShares(shareholder, share);
  }

  @Override @View public final BigInteger sharesOf(S shareholder) {
    return shares.getOrDefault(shareholder, ZERO);
  }

  @Override @FromContract(PayableContract.class) @Payable
  public void place(BigInteger amount, O offer) {
    require(offer.seller == caller(), "unauthorized");
    require(shares.containsKey(offer.seller), "unauthor.");
    require(sharesOf(offer.seller) - sharesOnSaleOf(offer.seller) >=
            offer.sharesOnSale, "not enough shares");
    offers.add(offer);
  }

  @Override @FromContract(PayableContract.class) @Payable
  public void accept(BigInteger amount, S buyer, O offer) {
    require(caller() == buyer, "unauthorized");
    require(offers.contains(offer), "unknown offer");
    require(offer.isOngoing(), "the offer is not ongoing");
    require(offer.cost <= amount, "not enough money");
    offers.remove(offer);
    removeShares(offer.seller, offer.sharesOnSale);
    addShares(buyer, offer.sharesOnSale);
    offer.seller.receive(offer.cost);
  }

  @Override @View public final BigInteger sharesOnSaleOf(S shareholder) {
    return offers.stream()
      .filter(o -> o.seller == shareholder && o.isOngoing())
      .map(offer -> offer.sharesOnSale)
      .reduce(ZERO, BigInteger::add);
  }
}
```

```
public interface Validators<V extends Validator> extends SharedEntity<V,Offer<V>> {

  @View BigInteger getStake(V validator);

  @FromContract @Payable
  void reward(String behaving, String misbehaving, BigInteger gasConsumed);
}

public abstract class AbstractValidators<V extends Validator>
    extends SimpleSharedEntity<V,Offer<V>> {

  public AbstractValidators(V validator, BigInteger power) {
    super(validator, power);
  }

  public void reward(String behaving, String misbehaving, BigInteger gasConsumed) { ... }
}

public class TendermintValidators extends AbstractValidators<TendermintED25519Validator> {

  public TendermintValidators(TendermintED25519validator validator, BigInteger power) {
    super(validator, power);
  }
}
```

Fig. 9 The shared entity of the validators set of a Hotmoka blockchain

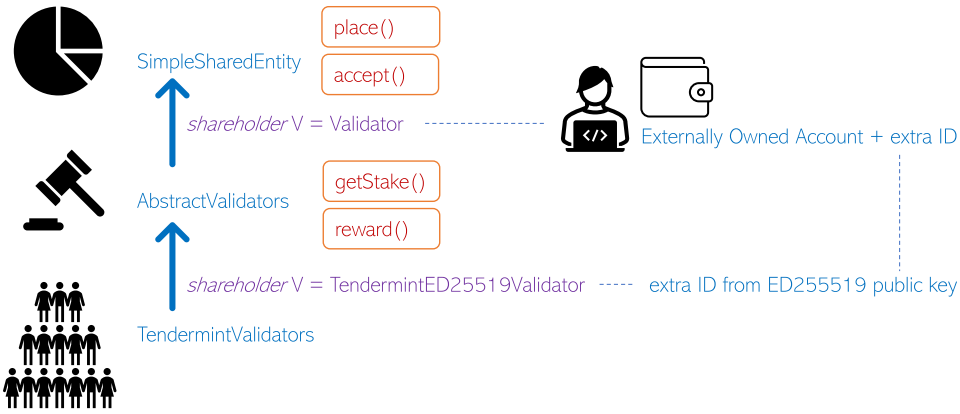

Fig. 10 Hierarchy of classes for implementing Hotmoka Validators

Since `SimpleSharedEntity` allows shares to be sold and bought, this holds for its `TendermintValidators` subclass as well: the set of validators is dynamic and it is possible to sell and buy voting power in order to invest in the blockchain and earn rewards at each block committed. At block creation time, Hotmoka calls method `getShareholders` inherited from `SimpleSharedEntity` and informs the underlying Tendermint engine about the identifiers of the validator nodes for the next blocks. Tendermint expects such validators to mine and vote the subsequent blocks, until a change in the validators' set occurs.

6 An attack to the shared entities contract

Let us state an important, expected property about shared entities:

Consistency of Shareholders

If `se` is a `SharedEntity<S,O>` object, then `se.getShareholders()` contains only elements of type `S`.

This property is important since it states that one can trust the type `S` of the shareholders: if one creates a `SharedEntity` and fixes a specific type `S` for its shareholders, then only instances of `S` will actually manage to become shareholders.

It turns out that the *Consistency of Shareholders* property holds for instances of the class `SimpleSharedEntity` in Fig. 8. Namely, that class does not use unchecked casts, hence it is strongly-typed [14] and its map `shares` actually holds values of type `S` in its domain, only. For this consistency result, one needs the dummy `buyer` argument for the method `accept` of the shared entities. Without that argument, the *Consistency of Shareholders* property would not hold, since one could only write `addShares((S) caller(), offer.sharesOnSale)` in the implementation of `accept` in Fig. 8, with an unchecked cast that makes its code non-strongly-typed. In that case, also contracts not of type `S` could call `accept` and become shareholders.

There is, however, a problem with the reasoning in the previous paragraph. Namely, absence of unchecked operations guarantees strong typing of Java *source* code. But what is installed and executed in blockchain is the Java bytecode that has been derived from the compilation of the code in Fig. 8. Malicious users might install in blockchain some manually crafted bytecode, not derived from its Java source code compiled together with the source code in Fig. 8. That crafted code might call the methods of `SimpleSharedEntitys` in order to attack that contract. In particular, the signature of method `accept` declares a parameter `buyer` of type `S` at source code level, but its compilation into Java bytecode declares an erased parameter `buyer` of type `PayableContract` instead. It follows that an attacker can install in blockchain a snippet of bytecode that calls `accept` and passes *any* `PayableContract`, not only those that are instances of `S`: the *Consistency of Shareholders* property is easily violated at bytecode level.

In particular, it is important that the *Consistency of Shareholders* property holds for the subclass `TendermintValidators`: its shareholders must be `TendermintED25519Validators` (as declared in the generic signature of `TendermintValidators` in Fig. 9) that enforce a match between their public key, that identifies who can spend the rewards sent to the validator, and their Tendermint identifier, that identifies which node of the blockchain must do the validation work (see how the constructor initializes `this.id` in Fig. 5). If it were possible to add a shareholder of another type `Attacker`, the code of `Attacker` could decouple the node identifier from its public key (see Fig. 11): Tendermint would expect

the node (belonging to the *victim*) to do the validation work while the owner of the private key of the `Attacker` could just wait for accrued rewards to spend. A sort of validator's slavery. Section 4 asserted that the *Consistency of Shareholders* property holds, at source level. Namely, an `attacker` (of type `Attacker`) can only become shareholder by accepting an ongoing sale `offer` of shares through a call to `tv.accept(offer.cost, attacker, offer)` (Fig. 8). This is impossible at source level (left part of Fig. 12), where that call does *not* compile, since `attacker` has type `Attacker` that is not an instance of `V`, which has been set to `TendermintED25519Validator`. But a Hotmoka blockchain contains only the bytecode of `SimpleSharedEntity`, where the signature of `accept` has been erased into `accept(BigInteger amount, PayableContract buyer, Offer offer)` (see Fig. 2 and the right part of Fig. 12). Hence a blockchain transaction that invokes `tv.accept(offer.cost, attacker, offer)` at bytecode level *does* succeed, since `attacker` is an externally owned account and all such accounts are instances of `PayableContract` (Fig. 3). That transaction adds `attacker` to the shareholders of `tv`, therefore violating the *Consistency of Shareholders* property and allowing validator's slavery.

7 A solution for fixing the compilation of the contract

The security issue in Sect. 6 is due to the over-permissive erasure of the signature of method `accept`, where the compiler gives `buyer` the type `PayableContract`. Therefore, a solution is to oblige the compiler to generate a more restrictive signature where, in particular, the parameter `buyer` has type `TendermintED25519Validator`: only that type of accounts must be accepted for the validators, consequently banning instances of `Attacker`.

The fixed code is shown in Fig. 13. The only difference is that method `accept` has been redefined to enforce the correct type for `buyer` (see that redefined method also in Fig. 3). For the rest, that method delegates to its implementation inherited from `AbstractValidators`, through a call to `super.accept`. It is important to investigate which is the Java bytecode generated from the code in Fig. 13. Since Java bytecode does not allow one to redefine a method and modify its argument types, the compiled bytecode actually contains *two* `accept` methods, as follows:

```
public class Attacker extends
    ExternallyOwnedAccount {

  public Validator(String publicKey) {
    super(publicKey);
  }

  @View
  public String id() {
    return /* id of the victim blockchain node,
        unrelated to publicKey */
  }
}
```

Fig. 11 An attacker that exploits the work of a blockchain validator node and fraudolently earns the rewards of that work

```
public class TendermintValidators extends
    AbstractValidators {
  ...

  public void accept(BigInteger,
      TendermintED25519Validator,Offer)
    aload_0
    aload_1
    aload_2
    aload_3
    invokespecial AbstractValidators.accept
        (BigInteger,PayableContract,Offer)
    return

  // synthetic bridge method
  public void accept(BigInteger,PayableContract,
      Offer)
    aload_0
    aload_1
    aload_2
    checkcast TendermintED25519Validator
    aload_3
    invokevirtual accept
        (BigInteger,
         TendermintED25519Validator,
         Offer)
    return
}
```

The first `accept` method above is the compilation of that from Fig. 13: it delegates to the `accept` method of the superclass `AbstractValidators`. The second `accept` method above is a *bridge method* that the compiler generates in order to guarantee that all calls to the erased signature `accept(BigInteger,PayableContract,Offer)` actually get forwarded to the first, redefined `accept`. It casts its `buyer` argument into `TendermintED25519Validator` and calls the first `accept`. This bridge method and its checked cast guarantee that only `TendermintED25519Validators` can become validators. As shown in Fig. 14, an instance of `Attacker` (Fig. 11) cannot be passed to the first `accept` (type mismatch) and makes the second `accept` fail with a class cast exception. The *Consistency of Shareholders* holds for instances of `TendermintValidators` now and the attack in Sect. 6 cannot occur anymore.

The solution of redefining method `accept` can be seen as a limited form of heterogeneous compilation of generics, restricted to a specific method and forced manually. It is interesting to consider which methods would need that

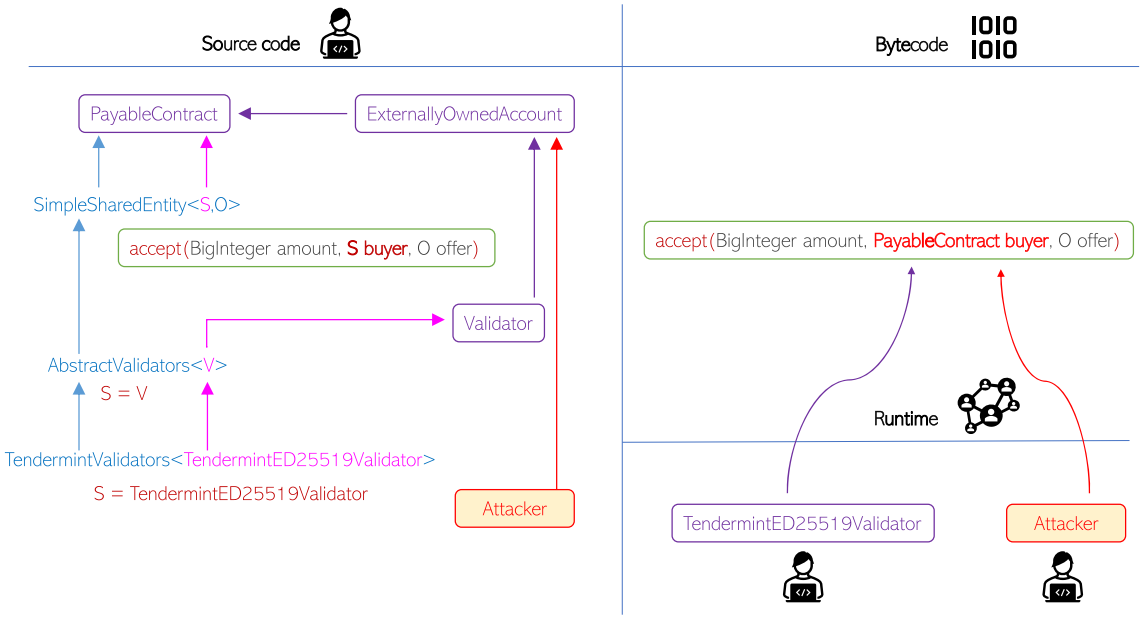

Fig. 12 Example of possible attack to a smart contract that uses Java generics

```
public class TendermintValidators extends AbstractValidators<TendermintED25519Validator> {

  public TendermintValidators(TendermintED25519validator validator, BigInteger power) {
    super(validator, power);
  }

  @Override @FromContract(PayableContract.class) @Payable
  public void accept
    (BigInteger amount, TendermintED25519Validator buyer, Offer<TendermintED25519Validator> offer) {
    super.accept(amount, buyer, offer);
  }
}
```

Fig. 13 The fixed code of the shared entity of the validators of a Hotmoka blockchain built over Tendermint

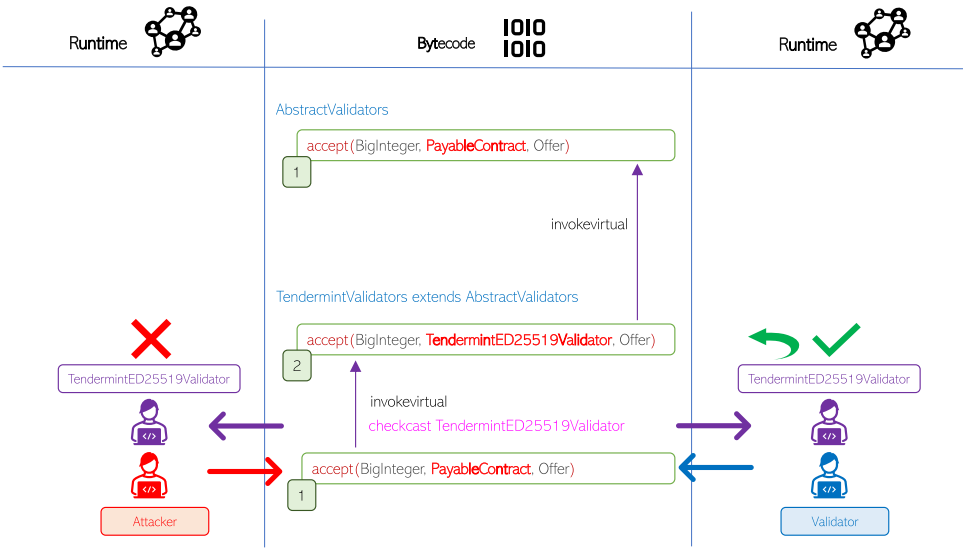

Fig. 14 Example of how the proposed solution works at run time when the `accept` method is called by a `Validator` or an `Attacker`

redefinition, in general. They are those that have a parameter of a generic type that is restricted in a subclass. For instance, method `accept` in Fig. 7 has parameters `buyer` and `offer` of generic type `S` and `O`, respectively. The subclass in Fig. 9 restricts `S` to be a `TendermintED25519Validator` and `O` to be an `Offer<TendermintED25519Validator>`. Hence one must redefine `accept` in the subclass with the more specific types for the `buyer` and `offer` parameters. In the future, a compiler might perform this automatically or a static analysis tool might issue a warning when such redefinition is needed. Currently, however, that is left to the programmer of the smart contracts, who might overlook the problem and give rise to security issues, as shown in Sect. 6.

8 Related Work

Programming languages specific to smart contracts (such as Solidity) do not have generic types. Conversely general-purpose programming languages do have generic types in most cases, but are much less frequently used for writing smart contracts. In any case, we are not aware of any scientific work on the use of generic types for writing more generic smart contracts, nor of any study on the security risks, and their solutions, that this implies for the resulting smart contracts. From this point of view, the present paper has no direct literature to compare with. Anyway, it is possible instead to insert this paper in the broader context of software correctness and security.

It has been estimated that, on average, software developers make from 100 to 150 errors for every thousand lines of code [20]. In 2002, the National Institute of Standards and Technology (NIST) estimates that the economic costs of faulty software in the US is about tens of billions of dollars per year and represent approximately just under one percent of the Nation's gross domestic product. The effects induced by errors in software development are even worse when such pieces of software are smart contracts. Indeed, it is usually impossible to change a smart contract once it has been deployed, the immutability being one of its main characteristics, so that errors are treated as intended behaviors. Moreover, smart contracts often store and manage critical data such as money, digital assets and identities. For this reason, smart contracts vulnerabilities and correctness are becoming important in literature [21]. Possible solutions can be classified into three main categories: (i) static analysis of EVM bytecode, (ii) automatic rectification of EVM bytecode and (iii) development of new languages for smart contracts.

Given the plurality of languages currently available for the design of smart contracts, static analysis is usually performed directly on the Ethereum bytecode, in order to make the solution general enough and promote its adoption. At this regards, SafeVM [22] is a verification tool for Ethereum smart contracts that works on bytecode and exploits the state-of-the-art verification engines already available for C programs. The basic idea is to take as input a smart contract in compiled bytecode, that can possibly contain some `assert` or `require` annotations, decompile it and convert it into a C program with `ERROR` annotations. This C program can be verified by using existing verification tools. In [23], the authors propose a verification tool for Ethereum smart contracts based on the use of the existing Isabelle/HOL tool, together with the specification of a formal logic for Ethereum bytecode. More specifically, the desired properties of the contracts are stated in pre/postcondition style, while the verification is done by recursively structuring contracts as a set of basic blocks down to the level of instructions. Another tool for the analysis of Ethereum bytecode is EthIR [24]. This open-source tool allows the precise decompilation into a high-level, rule-based representation. Given such representation, properties can be inferred through available state-of-the-art analysis tools for high-level languages. More specifically, EthIR relies on an extension of Oyente, a tool that generates code control-flow graphs in order to derive a rule-based representation of the bytecode. Considering the specific case of the Java language, formal techniques for static analysis can be built, for instance, over the Featherweight Java calculus [25], or by abstract interpretation [26]. Currently, however, we are not aware of formal verifications for generics, at bytecode level.

Relatively to the automatic certification of smart contracts, Solythesis [27] is a compilation tool for smart contracts that provides an expressive language for specifying desired safety invariants. Given a smart contract and a set of user defined invariants, it is able to produce a new enriched contract that will reject all transactions violating the invariants. Another solutions, based on bytecode rewriting, is presented in [28], where the authors propose the enforcement of security policies through the enhancement of bytecode. More specifically, the disassembled bytecode is instrumented through new security guard code that enforces the desired policy. Their initial efforts are mainly focused on the verification of arithmetic operations, such as the prevention of overflows. In the future, they plan to focus on verifying memory access operations. SMARTSHIELD [29] is another tool for automatically rectifying bytecode with the aim to fix three typical security bugs in smart contracts: (i) state changes after external calls, (ii) missing checks for out-of-bound arithmetic operations, and (iii) missing checks for failing external calls. More specifically, given an identified issue, the tool performs a semantic-preserving code transformation to

ensure that only the insecure code patterns are revised, eventually sending the rectification suggestions back to the developers when the eventual fixes can lead to side effects. The tool not only guarantees that the rectified contracts are immune to certain attacks but also that they are gas-friendly. Indeed, it adopts heuristics to optimize gas consumption.

The solution proposed in this paper could be implemented through an automatic bytecode rectification mechanism. Indeed, the additional method with a more restrictive signature could be automatically added in the bytecode without the need for an explicit method redefinition at the source code level.

Finally, as regards to the definition of new programming languages for safe smart contracts, Scilla [30] has been tailored by taking System F as a foundational calculus. It is able to provide strong safety guarantees by means of type soundness. Thanks to its minimalistic nature, it has been possible to define also a generic and extensible framework for lightweight verification of smart contracts by means of user-defined domain-specific analyses. The type variables of the functional foundational calculus can be seen as generic types. We do not know how they are compiled and if the strong typing guarantee of the source code extends to the compiled code as well. Scilla contracts are developed with the Neo Savant online IDE. Currently, neither Neo Savant IDE nor the block explorer allow one to inspect the compiled bytecode, in order to understand how generic types are compiled.

As regards to this last solution, which is based on the definition of new programming languages specific for writing safe smart contracts, the proposed solution could guide a more sophisticated and conscious bytecode generation. Indeed, the next generation of programming languages for smart contracts should take in mind that the generated bytecode should be called directly, without passing from the source code and the compiler checks. Therefore, any checks that are possible at source-code level, such as type checking, should remain possible also during bytecode execution.

9 Conclusion

This paper has shown that generics are useful in the definition of smart contracts and can simplify the development of rather complex code such as that for shared entities, and support code reuse, for instance to implement the validators set of a blockchain network. However, this paper has shown that generic types introduce risks of security as well. Namely, many programming languages, including Java, erase them at compile time into types that might be too permissive for low-level calls, such as those that are started by blockchain transactions. Note that the use of a programming language without generics is not the solution: Solidity has no generics and consequently erases *all* reference types into `address`. That is the worst possible erasure.

The solution in this paper has been to redefine the methods that have an argument of generic type, in such a way to call their superclass (see the case of `accept` in Fig. 13). This fixes the security risk, but cannot be regarded as the definite solution to the problem. It is just a trick that works because it forces the compiler to generate some specific kind of bytecode. A *smarter* compiler might recognize the redefined `accept` as *useless* and just remove it. This would recreate the issue that has been just solved. That is, the solution in this paper works only for the way compilers compile *today*.

With hindsight, it is questionable to have implemented generics by erasure and code instrumentation (bridge methods). If generics would be present and checked at bytecode level, the attack in Sect. 6 would just be impossible. Currently, generics can only exist as bytecode annotations that are not mandatory and are ignored by the Java virtual machine that runs the bytecode. The same consideration might be applied beyond generics: many features of modern programming languages have no direct low-level counterpart but are implemented via instrumentation. Examples are inner classes and closures (lambda expressions). This is fine at source level, but allows low-level calls to easily circumvent the encapsulation guarantees of the language. When embedded in a permissionless blockchain, such features become dangerous attack surfaces. This paper has shown the attack surface due to redefinition of methods with a generic parameter. But another example is the use of instrumented methods to allow access to private state from inner classes: since inner classes are compiled into distinct bytecode classes, the compiler adds non-private accessors to the private state. These accessors cannot be used at source level, but can be called at bytecode level to gain access to private state. This paper does not provide a solution to this other issue, but this further example makes it clear that the attack surface is larger than what described here.

Acknowledgements This work was partially supported by "Progetto di Eccellenza" of the Computer Science Department of University of Verona, Italy.

Funding Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement. The authors have not disclosed any funding.

Data availability The source code is freely available at https://github.com/Hotmoka/hotmoka/tree/master/io-takamaka-code

Declarations

Conflict of interest The authors declare that they have no conflicts of interest.

Ethical standard This article does not contain any studies involving animals performed by any of the authors.

Informed consent This article does not contain any studies involving human participants performed by any of the authors.

References

1. Casino, F., Dasaklis, T.K., Patsakis, C.: A systematic literature review of blockchain-based applications: current status. Classif. Open Issues Telemat. Inform. **36**, 55–81 (2019). https://doi.org/10.1016/j.tele.2018.11.006
2. Spoto, F.: A Java framework for smart contracts. In: 3rd Workshop on Trusted Smart Contracts (WTSC'19). Lecture Notes in Computer Science, vol. 11599, pp. 122–137. Springer, St. Kitts and Nevis (2019)
3. Spoto, F.: Enforcing determinism of Java smart contracts. In: 4th Workshop on Trusted Smart Contracts (WTSC'20). Lecture Notes in Computer Science, vol. 12063, pp. 568–583. Springer, Kota Kinabalu (2020)
4. Nakamoto, S.: Bitcoin: a Peer-to-Peer electronic cash system (2008). https://Bitcoin.org/Bitcoin.pdf.
5. Antonopoulos, A.M.: Mastering Bitcoin: Programming the Open Blockchain, 2nd edn. O'Reilly Media, Inc. (2017)
6. Buterin, V.: Ethereum whitepaper (2013). https://Ethereum.org/en/whitepaper/
7. Antonopoulos, A.M., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly (2018)
8. Migliorini, S., Gambini, M., Combi, C., La Rosa, M.: The rise of enforceable business processes from the hashes of blockchain-based smart contracts. In: Enterprise, Business-Process and Information Systems Modeling, pp. 130–138. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-20618-5_9
9. Crafa, S., Di Pirro, M., Zucca, E.: Is solidity solid enough? In: 3rd Workshop on Trusted Smart Contracts (WTSC'19). Lecture Notes in Computer Science, vol. 11599, pp. 138–153. Springer, St. Kitts and Nevis (2019)
10. Siegel, D.: Understanding the DAO attack (2016). https://www.coindesk.com/understanding-dao-hack-journalists
11. Cosmos: The internet of blockchains. https://cosmos.network
12. Hotmoka—blockchain and IoT with smart contracts in Java (2021). https://www.hotmoka.io
13. Hyperledger—open source blockchain technologies. https://www.hyperledger.org
14. Naftalin, M., Wadler, P.: Java generics and collections. O'Reilly Media (2006)
15. Benini, A., Gambini, M., Migliorini, S., Spoto, F.: Power and pitfalls of generic smart contracts. In: Third International Conference on Blockchain Computing and Applications (BCCA'21), IEEE, Tartu, Estonia, pp. 179–186 (2021). https://doi.org/10.1109/BCCA53669.2021.9657048
16. Odersky, M., Wadler, P.: Pizza into Java: translating theory into practice. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 146–159 (1997). https://doi.org/10.1145/263699.263715
17. Sallenave, O., Ducournau, R.: Lightweight generics in embedded systems through static analysis. SIGPLAN Not. **47**(5), 11–20 (2012). https://doi.org/10.1145/2345141.2248421
18. Spoto, F.: Hotmoka Github repository. GitHub Inc (2018–2022). https://github.com/Hotmoka/hotmoka
19. Kwon, J.: Tendermint: consensus without mining (2014). https://tendermint.com/static/docs/tendermint.pdf
20. Humphrey, W.S.: A Discipline for Software Engineering. Addison-Wesley Longman Publishing Co., Inc (1995)
21. Murray, Y., Anisi, D.A.: Survey of formal verification methods for smart contracts on blockchain. In: 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1–6 (2019). https://doi.org/10.1109/NTMS.2019.8763832
22. Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A.: SAFEVM: a safety verifier for Ethereum smart contracts. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 386–389 (2019). https://doi.org/10.1145/3293882.3338999
23. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL, pp. 66–77 (2018). https://doi.org/10.1145/3167084
24. Albert, E., Gordillo, P., Livshits, B., Rubio, A., Sergey, I.: EthIR: a framework for high-level analysis of Ethereum bytecode. In: Automated Technology for Verification and Analysis, pp. 513–520 (2018). https://doi.org/10.1007/978-3-030-01090-4_30
25. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. (TOPLAS) **23**(3), 396–450 (2001). https://doi.org/10.1145/503502.503505
26. Spoto, F.: The Julia static analyzer for Java. In: Proceedings of the 23rd Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 9837, pp. 39–57. Springer, Edinburgh (2016)
27. Li, A., Choi, J.A., Long, F.: Securing smart contract with runtime validation. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 438–453 (2020). https://doi.org/10.1145/3385412.3385982
28. Ayoade, G., Bauman, E., Khan, L., Hamlen, K.: Smart contract defense through bytecode rewriting. In: 2019 IEEE International Conference on Blockchain (Blockchain), pp. 384–389 (2019). https://doi.org/10.1109/Blockchain.2019.00059
29. Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S., Gu, D.: SMARTSHIELD: automatic smart contract protection made easy. In: IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 23–34 (2020). https://doi.org/10.1109/SANER48275.2020.9054825
30. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with Scilla. Proc. ACM Program. Lang. 3 (OOPSLA). https://doi.org/10.1145/3360611

Fausto Spoto is associate professor at the University of Verona, Italy, where he studies programming languages and software engineering. He has developed techniques for the static analysis of Java and Java bytecode. More recently, he studied the use of Java for writing smart contracts and developed the Hotmoka blockchain over Tendermint. His current research is in the area of static analysis and construction of verified smart contracts.

Sara Migliorini received the Master degree in Computer Science from the University of Verona in 2007 and the Ph.D. degree in computer science from the same university in 2012. From 2012 to 2019 she was a post-doc research associate at University of Verona and from 2019 she is assistant professor at the same university. Her main research interests include: geographic information systems, distributed architectures, big data systems and analytics, recommendation systems, and blockchain technology.

Mauro Gambini received the Master degree in Computer Science from the Università degli Studi di Verona in 2007 and the Ph.D. degree in computer science from the same university in 2012. From 2012 he is a post-doc research associate at Università degli Studi di Verona. Her main research interests include: programming languages, information systems and blockchain technology.

Andrea Benini graduated in Computer Science and Engineering at University of Verona in 2021. He is interested in blockchain and he contributed to the development of some projects such as Hotmoka, Commercio.network, Ethersolve. He is currently an IT Engineer and he works as a web developer in an IT consulting company.