Power and Pitfalls of Generic Smart Contracts

Andrea Benini
Dept. of Computer Science
University of Verona, Italy
andrea.benini@studenti.univr.it

Mauro Gambini
Dept. of Computer Science
University of Verona, Italy
mauro.gambini@univr.it

Sara Migliorini
Dept. of Computer Science
University of Verona, Italy
sara.migliorini@univr.it

Fausto Spoto
Dept. of Computer Science
University of Verona, Italy
fausto.spoto@univr.it

2021 Third International Conference on Blockchain Computing and Applications (BCCA) | 978-1-6654-2182-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/BCCA53669.2021.9657048

***Abstract*—Generics are a powerful feature of programming languages that allows one to write highly reusable code. More specifically, they are based on the use of type placeholders in order to produce parametrized code, that can be instantiated for each concrete type provided for them. In many programming languages, such as Java, they are implemented by *erasure, ie.* replaced by their upper bound type during compilation into bytecode. This paper originated from a real security issue that we found while using generics for writing smart contracts for blockchain, in order to implement a contract for *shared entities* (such as a company shared by its shareholders), for the Hotmoka blockchain, whose contracts are written in Java. The considered case study is particularly important since the validators' set of the blockchain itself is a special case of shared entities. The analysis shows that the power of generics comes at the risk of a too permissive typing of the compiled code, due to the erasure mechanism, with a consequent possible attack to the validators' set. This paper proposes a solution that forces the compiler to generate more precise type information than those arising by erasure.**

***Index Terms*—smart contract, generics, blockchain**

I. Introduction

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 the transactions or their ordering: a misbehaving single machine gets immediately put out of consensus and isolated. Bitcoin [1], [2] has been the first blockchain's success story. Here transactions are programmed in a non-Turing complete bytecode language, almost exclusively used to implement transfers of units of coins between *accounts*.

A few years after Bitcoin, another blockchain, called Ethereum [3], [4], introduced the possibility of programming transactions in an actual, imperative and Turing-complete programming language, called Solidity. Solidity's code is organized in *smart contracts*, that can be seen as objects that control money. A smart contract is essentially an agreement between two or more parties that can be automatically enforced without the need for a trustworthy intermediary [5]. Ethereum's transactions can hence execute much more than coin transfers. Namely, they run object constructors and methods, which results in a sort of *world computer* that persists the same objects inside the memory of all the computers composing the blockchain's network.

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 [6]. Since, at run time, it is the bytecode that gets executed, everything can be passed for `p`, not just a `Person` instance. 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 [7], that costed millions of dollars.

Strong typing is one of the reasons that push towards the adoption of *traditional* programming languages for smart contracts. For instance, the Cosmos blockchain [8] uses Go. The Hotmoka blockchain [9] uses a subset of Java for smart contracts, called Takamaka [10], [11]. Hyperledger [12] allows Go and Java. Another reason is the availability of modern language features, that are missing in Solidity, such as *generics, ie.* the possibility of using type variables. Generics are a powerful and very useful facility for programming smart contracts, since they allow one to personalize the behaviour of such contracts and partially overcome their inherent incompleteness [5]. In Java source code, generics are strongly typed, if no *unchecked operations* are used [13], as it will always be the case in this paper. However, generics might have security issue at the level of compiled Java code and this paper originated from a real issue that we found in our code.

The contribution of this paper is to show a real-life use of generics for an actual smart contract contained in the support library of the Takamaka language, and to demonstrate that a naïve use of Java generics can lead to a code security vulnerability which allows an attacker to earn money by exploiting someone else's work, with both economical and legal side effects. This paper will provide 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 bytecode languages for future smart contract languages, by learning from the weaknesses of Java bytecode.

The remainder of this paper is organized as follows. Sec.. II discusses the management of generics in Java. Sec. III introduces our real-life Java smart contract that uses generics. Sec. IV shows that a naïve deployment of a subclass of that contract leads to a code vulnerability. Sec. V presents a fix to that vulnerability. Sec. VI discusses some related work and, finally, Sec. VII concludes.

II. Generics Management in Java

There exists two common ways to implement generics in a programming language, that are often described in literature as *heterogeneous* and *homogeneous* [14]. 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 if the heterogeneous approach is the safest, it is rarely applied in particular in resource-constrained applications, because the code size may dramatically increase because of duplication [15]. 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. 1 and its method `accept`. The functionality of `SharedEntity` will be discussed later (Sec. III). Here, we only consider 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 we check the bytecode generated for `SharedEntity`, we 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.

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 [13] (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 class `SharedEntity`, as we will show later.

III. Shared Entities using Generics

A *shared entity* is 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 are the validator nodes of a proof of stake blockchain, where shares represent their voting power and remuneration percentage.

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

  @View
  BigInteger sharesOf(S shareholder);

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

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

  List<S> getShareholders();

  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. 1. A simplified part of our shared entity interface. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/dao/SharedEntity.java.

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 offers for selling shares.

We are developing for the Hotmoka blockchain, that uses smart contracts written in a subset of Java called Takamaka. Fig. 1 shows a simplification of our interface for shared entities. It includes an inner class `Offer` that models sale offers: it specifies who is the seller 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. By using class `Offer`, the `SharedEntity` interface specifies four methods. Method `sharesOf` allows one to know how many shares a potential `shareholder` (of type `S`) holds. It is annotated as `@View`. In Takamaka, this means that its execution can be performed *for free*, without paying gas, since it has no side-effects. Who wants to sell (some of) its shares calls method `place` with a sale `offer`. This method is annotated as `@Payable` since implementations are allowed to require a payment of `amount` coins for managing the sale. Who buys the 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 `amount` $\geq$ `offer.cost` coins to the seller. This means that shareholders must be able to receive payments and that is why `S extends PayableContract`: the `PayableContract`s, in Takamaka, are those that can receive payments. Method `getShareholders` yields the list of the current shareholders of the entity. It is not `@View`, since it creates a new list, which is a side-effect.

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`. In Takamaka, this could be written as `@FromContract(S.class)`. Unfortunately, Java does not allow a generic type variable `S` in `S.class`. Due to this syntactical limitation of Java, the best we could write in Fig. 1 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 our abstraction, we will have to program explicit dynamic checks in code, as shown later, and this will be the reason of the parameter `buyer` in `accept`.

Let us state an important property about shared entities:

> *Consistency of Shareholders*
>
> If `se` is a `SharedEntity<S,O>` then the elements contained in the list `se.getShareholders()` have type `S`.

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

Fig. 2 shows our implementation of the `SharedEntity` interface in Fig. 1 by using two fields: `shares` maps each shareholder to its amount of shares and `offers` collects the offers that have been placed. The constructor initially populates the map `shares`. 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`, like `offer.seller`. As we 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

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

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

  public SimpleSharedEntity
    (S[] shareholders, BigInteger[] shares) {
    require(shareholders.length == shares.length,
            "length mismatch");
    for (int pos = 0; pos < shareholders.length; pos++)
      addShares(shareholders[pos], shares[pos]);
  }

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

  @Override @FromContract(PayableContract.class) @Payable
  public void place(BigInteger amount, O offer) {
    require(offer.seller == caller(),
            "not authorized to sell");
    require(shares.containsKey(offer.seller),
            "only shareholders can sell");
    require(sharesOf(offer.seller)
        .subtract(sharesOnSaleOf(offer.seller))
        .compareTo(offer.sharesOnSale) >= 0,
        "not enough shares to sell");
    offers.add(offer);
  }

  @Override @FromContract(PayableContract.class) @Payable
  public void accept(BigInteger amount, S buyer, O offer) {
    require(caller() == buyer,
            "only the future owner can buy the shares");
    require(offers.contains(offer), "unknown offer");
    require(offer.isOngoing(),
            "the sale offer is not ongoing anymore");
    require(offer.cost.compareTo(amount) <= 0,
            "not enough money");
    offers.remove(offer);
    removeShares(offer.seller, offer.sharesOnSale);
    addShares(buyer, offer.sharesOnSale);
    offer.seller.receive(offer.cost);
  }

  private BigInteger sharesOnSaleOf(S shareholder) {
    return offers.stream()
      .filter(offer -> offer.seller == shareholder &&
              offer.isOngoing())
      .map(offer -> offer.sharesOnSale)
      .reduce(ZERO, BigInteger::add);
  }

  @Override
  public List<S> getShareholders() {
    return shares.keyList();
  }
}
```

Fig. 2. A simplified part of our implementation of the shared entity interface. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/dao/SimpleSharedEntity.java.

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 we 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. 2) and the seller of the `offer` receives the required price `offer.cost`. Method `getShareholders()` yields the list of the keys in the domain of map `shares`.

It turns out that the *Consistency of Shareholders* property holds if Java source code creates and populates `SimpleSharedEntity`s. Namely, the code in Fig. 2 does not use unchecked casts, hence it is strongly-typed [13] and the map `shares` actually holds values of type `S` in its domain, only. For this consistency result, we had to pay the price of the dummy `buyer` argument for method `accept`. Without that argument, *Consistency of Shareholders* would not hold, since we could only write `addShares((S) caller(), offer.sharesOnSale)` in the implementation of `accept` in Fig. 2, with an unchecked cast that makes the 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. 2. 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. 2. That crafted code might call the methods of `SimpleSharedEntity`s 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 general, the problem arises whenever a method declares a formal parameter of generic type, such as `buyer` in method `accept` of Fig. 1. Next section shows the actual significance of the consequent security risk.

IV. An Attack to the Shared Entities Contract

This paper originated from an actual security issue that we found in our code that used shared entities in order to model the validators of a blockchain. Namely, the Hotmoka blockchain is built over Tendermint [16], a generic engine for replicating an application over a network of nodes. In our case, the application is the executor of smart contracts in Java, such as that in Fig. 2. 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

```
public abstract class Validator
       extends ExternallyOwnedAccount {

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

  @View
  public abstract String id();
}

public final class TendermintED25519Validator
       extends Validator {

  // the first 40 characters of the hexadecimal
       representation
  // of the sha256 hashing of the public key bytes
  private final String id;

  public TendermintED25519Validator(String publicKey) {
    super(publicKey);
    this.id = /* derived from publicKey as in the comment
         for id above */
  }

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

Fig. 3. The representation, in Hotmoka, of a validator of a Hotmoka blockchain based on Tendermint. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/governance/tendermint/TendermintED25519Validator.java.

and voting their acceptance. In general, 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. 3). 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 `AbstractValidators` class implements the validators' set and the distribution of the reward and is a subclass of `SimpleSharedEntity` (see Fig. 4). Shares are voting power in this case. Its subclass `TendermintValidators` restricts the type `S` of the shareholders to `TendermintED25519Validator`. At each block committed, Hotmoka calls the `reward` method of `AbstractValidators` in order to reward the validators that behaved correctly and punish 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.

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

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

  public AbstractValidators
    (V[] validators, BigInteger[] powers) {
    super(validators, powers);
  }

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

public class TendermintValidators
    extends AbstractValidators<TendermintED25519Validator>
         {

  public TendermintValidators
       (TendermintED25519validator[] validators,
        BigInteger[] powers) {
    super(validators, powers);
  }
}
```

Fig. 4. The shared entity of the validators of a Hotmoka blockchain. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/governance/AbstractValidators.java.

blocks. Tendermint expects such validators to mine and vote the subsequent blocks, until a change in the validators' set occurs.

It is important that the *Consistency of Shareholders* property holds for `TendermintValidators`: its shareholders must be `TendermintED25519Validator`s (as declared in the generic signature of `TendermintValidators`) 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. 3). 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. 5): 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. Sec. III 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)` (see Fig. 2). This is impossible at source level, where that call does *not* compile, since `attacker` has type `Attacker` that is not an instance of `S`, 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)`. 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`. That transaction adds

```
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. 5. An attacker that exploits a blockchain node for validation and fraudolently earns the rewards of the work of that node.

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

  public TendermintValidators
       (TendermintED25519validator[] validators,
        BigInteger[] powers) {
    super(validators, powers);
  }

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

Fig. 6. The shared entity of the validators of a Hotmoka blockchain built over Tendermint. Its full code is available at https://github.com/Hotmoka/hotmoka/blob/master/io-takamaka-code/src/main/java/io/takamaka/code/governance/tendermint/TendermintValidators.java.

`attacker` to the shareholders of `tv`, therefore violating the *Consistency of Shareholders* property and allowing validator's slavery.

V. Fixing the Compilation of the Contract

The security issue in Sec. IV is due to the over-permissive erasure of the signature of method `accept`, where `buyer` is given 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. 6. The only difference is that method `accept` has been redefined to enforce the correct type for `buyer`. 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. 6. 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 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. 6: 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 `TendermintED25519Validator`s can become validators. For instance, an instance of `Attacker` (Fig. 5) 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 Sec. IV 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 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. 1 has parameters `buyer` and `offer` of generic type `S` and `O`, respectively. The subclass in Fig. 4 restricts `S` to be a `TendermintED25519Validator` and `O` to be an `Offer<TendermintED25519Validator>`. Hence we 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, this is left to the programmer of the smart contracts, who might overlook the issue and give rise to security issues, as shown in Sec. IV.

VI. Related Work

It has been estimated that, on average, software developers make from 100 to 150 errors for every thousand lines of code [17]. 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 1 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 behaviours. 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 [18]. 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 [19] 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 [20], 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 [21]. 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 [22], or by abstract interpretation [23]. Currently, however, we are not aware of formal verifications for generics, at bytecode level.

Relatively to the automatic certification of smart contracts, Solythesis [24] 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 [25], 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 [26] 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.

Finally, as regards to the definition of new programming languages for safe smart contracs, Scilla [27] 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.

VII. Conclusion

This paper has shown that generic types can be useful in the definition of smart contracts, but introduce risks of security since in many programming languages, including Java, they get erased at compile time into types that might be too permissive for low-level calls, such as those that start blockchain transactions. Note that using 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 a method with an argument of generic type in such a way to call its superclass (see `accept` in Fig. 6). This fixes the security risk, but cannot be regarded as an elegant solution. It is just a trick that works because it forces the compiler to generate some kind of bytecode. A *smarter* compiler might recognize the redefined `accept` as *useless* and just remove it. This would recreate the issue that we have just solved. That is, our solution 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 Sec. IV 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, from inner classes to closures. 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.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008, available at https://bitcoin.org/bitcoin.pdf.

[2] A. M. Antonopoulos, *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed. O'Reilly Media, Inc., 2017.

[3] V. Buterin, "Ethereum Whitepaper," 2013, available at https://ethereum.org/en/whitepaper/.

[4] A. M. Antonopoulos and G. Wood, *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly, 2018.

[5] S. Migliorini, M. Gambini, C. Combi, and M. La Rosa, "The Rise of Enforceable Business Processes from the Hashes of Blockchain-Based Smart Contracts," in *Enterprise, Business-Process and Information Systems Modeling*. Springer International Publishing, 2019, pp. 130–138.

[6] S. Crafa, M. Di Pirro, and E. Zucca, "Is Solidity Solid Enough?" in *3rd Wokshop on Trusted Smart Contracts (WTSC'19)*, ser. Lecture Notes in Computer Science, vol. 11599. St. Kitts and Nevis: Springer, 2019, pp. 138–153.

[7] D. Siegel, "Understanding the DAO Attack," https://www.coindesk.com/understanding-dao-hack-journalists, June 2016.

[8] "Cosmos: The Internet of Blockchains," https://cosmos.network.

[9] "Hotmoka – Blockchain and IoT with Smart Contracts in Java," Available at https://www.hotmoka.io, 2021.

[10] F. Spoto, "A Java Framework for Smart Contracts," in *3rd Wokshop on Trusted Smart Contracts (WTSC'19)*, ser. Lecture Notes in Computer Science, vol. 11599. St. Kitts and Nevis: Springer, February 2019, pp. 122–137.

[11] ——, "Enforcing Determinism of Java Smart Contracts," in *4th Wokshop on Trusted Smart Contracts (WTSC'20)*, ser. Lecture Notes in Computer Science, vol. 12063. Kota Kinabalu, Malaysia: Springer, February 2020, pp. 568–583.

[12] "Hyperledger – Open Source Blockchain Technologies," https://www.hyperledger.org.

[13] M. Naftalin and P. Wadler, *Java Generics and Collections*. O'Reilly Media, 2006.

[14] M. Odersky and P. Wadler, "Pizza into Java: Translating Theory into Practice," in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997, pp. 146–159. [Online]. Available: https://doi.org/10.1145/263699.263715

[15] O. Sallenave and R. Ducournau, "Lightweight Generics in Embedded Systems through Static Analysis," *SIGPLAN Not.*, vol. 47, no. 5, pp. 11–20, 2012. [Online]. Available: https://doi.org/10.1145/2345141.2248421

[16] J. Kwon, "Tendermint: Consensus without Mining," 2014, https://tendermint.com/static/docs/tendermint.pdf.

[17] W. S. Humphrey, *A Discipline for Software Engineering*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[18] Y. Murray and D. A. Anisi, "Survey of Formal Verification Methods for Smart Contracts on Blockchain," in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2019, pp. 1–6.

[19] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "SAFEVM: A Safety Verifier for Ethereum Smart Contracts," in *Proc. of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 386–389. [Online]. Available: https://doi.org/10.1145/3293882.3338999

[20] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL," 2018, pp. 66–77. [Online]. Available: https://doi.org/10.1145/3167084

[21] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode," in *Automated Technology for Verification and Analysis*, 2018, pp. 513–520.

[22] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A Minimal Core Calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 3, pp. 396–450, 2001.

[23] F. Spoto, "The Julia Static Analyzer for Java," in *Proc. of the 23rd Static Analysis Symposium (SAS)*, ser. Lecture Notes in Computer Science, vol. 9837. Edinburgh, UK: Springer, September 2016, pp. 39–57.

[24] A. Li, J. A. Choi, and F. Long, "Securing Smart Contract with Runtime Validation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 438–453. [Online]. Available: https://doi.org/10.1145/3385412.3385982

[25] G. Ayoade, E. Bauman, L. Khan, and K. Hamlen, "Smart Contract Defense through Bytecode Rewriting," in *2019 IEEE International Conference on Blockchain (Blockchain)*, 2019, pp. 384–389.

[26] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "SMARTSHIELD: Automatic Smart Contract Protection Made Easy," in *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 23–34.

[27] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer Smart Contract Programming with Scilla," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360611