SmartInspect: Smart Contract Inspection Technical Report

Santiago Bragagnolo, Henrique S C Rocha, Marcus Denker, Stéphane Ducasse

SmartInspect: Smart Contract Inspection Technical Report

Santiago Bragagnolo(Inria) Henrique Rocha (Inria)
Marcus Denker (Inria) Stéphane Ducasse (Inria)

December 21, 2017

Abstract

Smart contracts are embedded procedures stored with the data they act upon. Debugging deployed Smart Contracts is a difficult task since once deployed, the code cannot be reexecuted and inspecting a simple attribute is not easily possible because data is encoded. In this technical report, we present SmartInspect to address the lack of inspectability of a deployed contract. Our solution analyses the contract state by using decompilation techniques and a mirror-based architecture to represent the object responsible for interpreting the contract state. SmartInspect allows developers and also end-users of a contract to better visualize and understand the contract stored state without needing to redeploy, nor develop any ad-hoc code.

1 Introduction

Blockchain technology has attracted a lot attention recently [14]. A blockchain is a distributed database, managed by a peer-to-peer network that stores a list of blocks or records. Ethereum [7], and BitCoin [15] are examples of blockchain technologies. Blockchains can be used for many applications such as cryptocurrency, digital wallets, adhoc networks, remote transactions, among other uses [5–7, 10, 11, 14, 15]. One notable application of blockchain is the execution of smart contracts [13].

Smart contracts are what embedded procedures are for databases: programs executed in the blockchain to manage and transfer digital assets. When used in platforms like Ethereum, the contract language is Turing-complete [1]. Therefore, smart contracts can be used in many different scenarios. For example, there are smart contracts employed to subcurrency [8], and outsourced computation [14]. Solidity [8] is the predominant programming language used to specify smart contracts on the Ethereum blockchain platform.

Smart contracts define data structure as well as the operations used to interact with this data [8]. Far from a typical database, where the primary representation is data, and the

available operations are about the structure and the content, the principal element of a Ethereum database is not just the data. In addition the database stores the behavior provided to interact with this data, and to trigger other behaviors, by sending messages to other contracts. Ethereum is a database that works as a stored environment of contract instance (objects). Compiled versions of the contract instances are then published as part of transactions to the blockchain. Blocks are chained together via hashes and build together the blockchain.

One of the challenges faced by developers of smart contracts is finding and fixing bugs. Indeed, contracts are opaque in the sense that once deployed in the blockchain it is difficult to access the value of a given contract attribute. In this report, we focus on inspecting a smart contract state as a first step to support contract debugging.

The difficulty to inspect contract data is not a widely known problem. Although there are many tools for traditional databases to access its stored data, as far as we know, there is no such tool to inspect contract information[1]. On the other hand, there are two practices that we can use to access contract data: (i) introducing getter methods, which requires the redeployment of the contract (if it is already running in the blockchain) and a possible data conversion (if the type is not supported as return); and (ii) using the API to acquire raw data and applying an ad-hoc decoding of the content. Both practices are tedious time-consuming tasks for a developer. From a business perspective, companies could use contract inspection to help clients better understand the information that is actually stored in the contract. In fact, the UTOCAT[2] company deemed this interaction with clients as an important scenario, regarding the complexity to explain and understand Ethereum technology possibilities.

As a solution, we propose SmartInspect, an inspector based on pluggable property reflection. The main idea is that the binary structure of contract is decompiled using a memory layout reification. The memory layout reification is built from the Solidity source code. Our SmartInspect architecture is based on decompilation capabilities encapsulated in mirrors [2]. Such mirrors are automatically generated from an analysis of Solidity source code. This approach allows us to access unstructured information from a deployed contract in a structured way.

The remainder of this tech report is organized as follows. Section 2 starts with a example of a smart contract. In Section 3, we detail the main problem our proposed approach aims to address. Section 4 describes our proposed solution, SmartInpect. Section 5 evaluates SmartInspect by comparing to other practices and also showing developers feedback. Section 6 provides a brief discussion on the inspection. In Section 7, we describe the related work. Finally, Section 8 presents our conclusions and outlines possible future work ideas.

[1] When the first version of this technical report and the SmartInspect tool became available, there were no other inspection solution available. Now we are aware of new inspection tools that we will evaluate on a future research paper.

[2] https://www.utocat.com/en/, verified 2017-10-20.

2 Smart Contract by Example

In this section, we present an example of a smart contract written in Solidity (Section 2.1), and we also describe a client application in Pharo Smalltalk to interact with it (Section 2.2). We use this example throughout the report to explain the opacity problem.

2.1 Poll Smart Contract

In this example, the contract manages a poll where users are allowed to vote a single time. Only the contract owner is allowed to modify the list of voters. The poll is managed with a contract because it is used for management decisions that rely on the veracity of the information.

The following listing contains the code of this contract:

Listing 1: Solidity Poll Contract Example

```
1  pragma solidity ^0.4.16;
2
3  contract Public3StatesPoll {
4    /* Type Definition */
5     enum Choice { POSITIVE, NEGATIVE, NEUTRAL }
6     struct PollEntry { address user; Choice choice; bool hasVoted; }
7
8    /* Properties */
9     PollEntry[] pollTable;
10    address owner;
```

This contract defines two user types Choice (line 5) and PollEntry (line 6). A Choice models the answers to the poll (whether the vote was *positive*, *negative* or *neutral*). A PollEntry is a record representing a vote, *i.e.,* the vote user, the selected option, and if he has voted or not. Note that to refer to the user we need an account address (using the primitive type address) that refers to an external account.

The contract stores internally a poll table (an array of PollEntry) (line 9) and an address to the contract's owner account (line 10). The poll table is an empty array where the contract owner will eventually store the poll information (i.e., the array will have an entry for each user that is allowed to vote). The contract owner's address is used for security checks.

```
11     /* Constructor */
12    function Public3StatesPoll () {
13       owner = msg.sender;
14    }
```

Lines 11-14 define the contract constructor. This constructor is executed when the contract is deployed in the blockchain. It keeps track of the user who owns the smart contract for future reference.

```
15    function isRegistered (address voterAccount) returns (bool) {
16       return (voterIndex (voterAccount) > -1);
17    }
18    function voterIndex (address voterAccount) returns (int) {
19       for (uint x = 0; x < pollTable.length; x++) {
20          if (pollTable[x].user == voterAccount) {
21             return int(x);
22          }
23       }
24       return -1;
25    }
```

We define the helper function voterIndex (lines 18-25), which returns the index of the voter in the poll table. We also created the function isRegistered (lines 15-17) to determine whether the user was registered to vote by using the voterIndex function. Since array indexes in Solidity are unsigned integers (uint), we need to explicitly convert it to a regular integer (line 21).

```
26    function addVoter(address voterAccount) returns (uint) {
27       assert( owner == msg.sender );
28       assert( !isRegistered(voterAccount) );
29       pollTable.push(PollEntry(voterAccount, Choice.NEUTRAL,
             false));
30       return pollTable.length -1;
31    }
32    function vote (Choice choice) {
33       assert( isRegistered(msg.sender) );
34       uint index = uint(voterIndex(msg.sender));
35       assert( !pollTable[index].hasVoted );
36       pollTable[index].choice = choice;
37       pollTable[index].hasVoted = true;
38    }
39    function votesFor(Choice choice) returns (uint) {
40       uint votes = 0;
41       for (uint x = 0; x < pollTable.length; x++) {
42          if (pollTable[x].hasVoted && pollTable[x].choice ==
               choice)
43             votes = votes +1;
44       }
45       return votes;
46    }
47    function allParticipantsHaveVoted () returns (bool) {
48       for(uint x = 0; x < pollTable.length; x++) {
49          if (!pollTable[x].hasVoted) return false;
50       }
51       return true;
52    }
53 } //end of contract
```

The rest of the contract defines the following functions:

- *addVoter* (lines 26-31). This function registers a voter into the poll table. It tries to assert[3] that the caller is the contract owner and the voter is not already registered.

[3]The *assert* function checks for a condition and throws an exception if such condition is not met. In

- *vote* (lines 32-38). This function assigns the given choice to the entry related to the calling user. The user must be registered and not voted yet.
- *votesFor* (lines 39-46). It returns the number of users that voted for the given choice.
- *allParticipantsHaveVoted* (lines 47-52). It returns true if all the registered users have voted.

2.2 Client Side

Once our poll contract is deployed in the blockchain, we need a client application to interact with it. For example, we can implement a web application providing a user interface or a web service as a means to invoke the functions in the contract and vote or get the poll results. The following listing illustrates the code of a Poll class implemented in Pharo for our client application that will act as a façade to our contract:

Listing 2: Client side for the Voter Smart Contract

```
1 Object subclass: #Poll
2  instanceVariableNames: 'deployedContract'
3  package: #PollContract.
```

Lines 1-3 declare a Poll class with a deployedContract instance variable. This instance variable refers to a proxy to the deployed contract. This is a common implementation used in other Ethereum client.

```
4 Poll class>> config
5  ^{#fromAccount →self systemAccount.
6    #gas →30000. #etc}.
7
8 Poll class>> deployNewContract: src
9       accounts: accounts
10      connection: conn
11
12  deployedContract := conn deploy: source
13                  configuration: self config.
14  accounts do: [ :account |
15    deployedContract addVoter: account configuration: self config ].
```

The class method[4] deployNewContract:accounts:connection: (lines 8-15) receives the contract source code, a list of user accounts that are allowed to vote and a connection to the blockchain. It first deploys the contract in the blockchain using the contract source code, and then calls the addVoter:configuration: function of the new contract for each of the given user accounts. When we call functions from a deployed contract, we need to provide configuration information as well.

Solidity, exceptions undo all changes made in the invoked method.

[4]A class method is comparable to a static method in the Java jargon. In Pharo method call with multiple parameters place arguments in between the method name. Hence this.foobar(arg1, arg2) is expressed as this foo: arg1 bar: arg2.

```
16 Poll>> config: usr
17  ^ {#fromAccount →usr account.
18    #gas →30000. #etc}.
19
20 Poll>> user: usr votes: aValue
21  deployedContract vote: aValue configuration: (self config: usr).
22
23 Poll>> isFinished
24  ^ deployedContract allParticipantsHaveVoted configuration: (self
      class config).
25
26 Poll>> results
27  ^ { #POSITIVE . #NEGATIVE . #NEUTRAL } collect: [ :value | value
      →deployedContract votesFor: value configuration: (self class
      config) ].
```

The method config: (lines 16-18) provides configuration data with the user's account. The method user:votes: (lines 20-22) invokes the function vote:configuration: from the contract using the user's configuration. Likewise, the method isFinished (lines 23-25) invokes the function allParticipantsHaveVoted() of the contract. The method results (lines 26-28) invokes repeatedly votesFor() for each of the contract choices and returns a map relating each choice to the number of people that voted for it.

It is noteworthy that the Poll class works as a thin layer over the remote contract performing remote calls to it. In the scenario of a real business application, this layer may define the complete process of a large business and its logic.

3 The Problem: Contract opaqueness

Contrary to traditional SQL databases such as Oracle or PostgreSQL which have a multitude of tools (e.g., DBeaver, Navicat, Sql Maestro, Toad, PgAdmin, etc.) to access the database schema and the *actual* data stored in a given column or row, Ethereum/-Solidity does not provide any tool to inspect contract state in the referential model of the application. Since the contract is an arbitrary data type, the offered API to interact and inspect is both restricted and at a low-level of abstraction.

Contract state is *read-only* in the sense that unauthorized clients cannot interact with it. Finally, contract state is *opaque*: since it is encoded there is no simple way for a software developer to know the actual value of a contract specific attribute effectively stored in the blockchain. The remote architecture of deployed solutions should also be taken into account.

3.1 Contract Remote Structure

A contract is stored in the blockchain database and its object representation can be accessed in the application client layer. The Ethereum platform employs proxies based

on the contract ABI[5] for covering the gap introduced by the physical location of the object (which is stored remotely in the blockchain). For example, when we invoke the method vote: configuration: in a Pharo client, the method call will be sent to a proxy that will connect through RPC (Remote Procedure Call) to the remote ABI object. The result of this method is a transaction receipt hash and, if applicable, the client will also receive any returned values of the method call. It is noteworthy that method calls to blockchain objects may have a transaction cost related with them.

By proxying the remote contract, a client application can use the contract methods just as any other object. Moreover, the client can activate methods that will be executed elsewhere in the blockchain (by paying the transaction cost if applicable) and it will return values that we can use (as any other values for other methods and objects). This simple way to interact with the contract is unsatisfactory, since the objects cannot be inspected. Since there is no simple way to access contract properties, it makes debugging session tedious or even impossible.

3.2 Opaqueness problem example

We use the poll contract example (Section 2) to illustrate the opaqueness problem. Let's suppose that a new user arrives a couple of days before the poll expiration date. When he tries to vote, the client system executes a routine that calls the contract's vote function and reports whether the call was successful (Listing 3). More specifically, the routine calls our method user: votes: defined in the Poll class (Listing 2, lines 16-17) that uses a proxy to remote call the vote function in the deployed contract.

Listing 3: Client voting routine

```
1 UserSession >> vote: aValue
2  transactionReceipt := poll user: user votes: aValue.
3  transactionReceipt
4    onSuccess: [ :t | self informToUser ];
5    onError: [ :e | self informError: e ].
```

The call will fail because the user was not a registered voter. We can see in the client code that the only point where there is a setup of users in the contract is during its deployment (Listing 2, lines 8-15). Therefore, the contract will not encounter the user and it will throw an exception. In the client, the details that caused the exception will be hidden, it will only know that the invoked method failed. Moreover, since the error is being thrown by the remote object, inspecting the contract code could identify the problem. However, a regular user does not have access to the contract code, only the people in charge of the contract have such access. For this reason, the user's only option is to submit a bug report stating that he cannot vote.

The person in charge (let's call him Bob) of solving this issue, will go and check the contract code (Listing 4), and deduce that there are be two possible reasons for the code

[5]Application Binary Interface (ABI) is the Ethereum standard to interact with contracts. This standard encodes contract data according to its specification.

to fail: (i) the user is not authorized, i.e., he/she was not registered into the contract; or (ii) the user already voted.

Listing 4: Contract vote function highlithing the asserts

```
1  function vote (Choice choice) {
2     assert( isRegistered(msg.sender) );
3     uint index = uint(voterIndex(msg.sender));
4     assert( !pollTable[index].hasVoted );
5     pollTable[index].choice = choice;
6     pollTable[index].hasVoted = true;
7  }
```

However, Bob cannot know for certain what caused the issue without analyzing the contract data. In this specific case, since we know the problem was caused by the unregistered voter, the easiest solution would be to change the contract state manually by calling the function addVoter to add the new user.

Bob will face many difficulties to find the issue. As we can see, we did not define the contract properties as public (Listing 1, lines 9-10). Therefore, if Bob wants to find the nature of the error, he will need the contract instance's current state to inspect it. Bob has two possibilities then: (i) re-instantiate the contract to add a new function to return the data (i.e., create a getter method); or (ii) develop a costly, ad-hoc decoder for reading the binary content of the contract.

If Bob takes the first possibility, he will add a new function to analyze the content of the contract. Since Solidity functions cannot return arrays or structs, Bob will need to adapt its function accordingly to acquire the poll data. Moreover, the contract will have to be redeployed, creating a new instance of it. Therefore, Bob will be able to analyze the new instance data with his function, but not the previous one (which was the one that presented the issue). Besides, there are the transactional costs to redeploy the contract to be considered, as well as the inconvenience to ask the users to vote again (since it is a new instance). In our example, where there is only a few days left to close the poll, it would not be feasible for Bob to ask all users to vote again.

The second possibility is for Bob to spend time into creating an add-hoc decoder. The main advantage on this possibility is that Bob does not need to redeploy the contract. The decoder could access the complex binary slots of the contract's related storage and converted them into the desired content that Bob is trying to analyze. Since the Solidity documentation for its binary enconding is incomplete, Bob will have a difficult time to create the decoder. Moreover, this decoder is a one time solution, as it is designed for a specific data in a particular contract, i.e., Bob will not be able to easily reuse this solution to another contract. In our example, Bob might not have the time required to create such decoder before the poll expiration date.

This scenario illustrates a simple aspect of the impact of the opacity of contracts. The general concern is that the developer should be able to understand the value of a given contract attribute.

3.3 Challenges

There are many challenges to pierce through the opaqueness problem and reveal contract information.

- **Binary and incomplete specification.** From the technical aspects we only have the Ethereum API to access a binary representation of the contract. The first challenge we faced is an *incomplete* specification of the contract encoding performed by the Solidity compiler [8].
- **Inconsistent specification of hash computation.** Another challenge related to the specification is the hash computation for dynamic types. The hash computation shows issues as it is not consistent with the specification.
- **Packed and ordered data.** We also highlight the challenges on decoding types, as the compiler packs as much data as possible into contiguous memory. Therefore, we need to know the specific types in the correct order to acquire the contract data, and that is not an easy task when we have an incomplete specification.

We acknowledge as a problem the challenges and difficulties of analyzing our own objects which are deployed in the Ethereum blockchain database. To solve this problem we propose an inspector that allows the user to perceive a clean representation of the object he is dealing with.

4 SmartInspect: Contract Inspector

SmartInspect is a local pluggable mirror-based reflection system for remotely deployed objects on a reflection-less system (the Ethereum platform). The goal of SmartInspect is to allow the inspection of known contracts based on its source code focusing on the debugging properties of interactiveness and distribution [16]. This reflective approach allows a user to see the contents of any contract instance of the given source code, without needing to redeploy, nor develop any ad-hoc code.

SmartInspect is implemeted in Pharo and it is publicly available as part of the SmartShackle tool suite.[6]

4.1 The Basics

The general idea of the Smart Inspector is to decompile the storage layout encoded by the Ethereum API (Figure 1). The decompilation employs a local pluggable mirror-based reflection architecture for remotely deployed objects on an Ethereum network.

Figure 2 shows the process to inspect contract, the pluggable reflective architecture generates a mirror for a given contract's source code by using its AST (Abstract Syntax

[6] https://github.com/RMODINRIA-Blockchain/SmartShackle, verified 2017-10-30

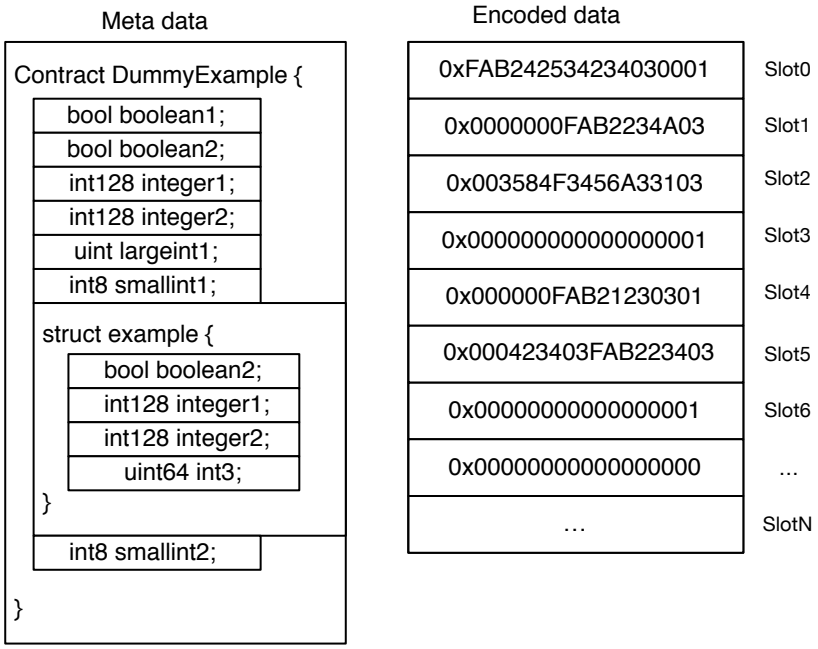

Figure 1: Static storage with the related code definition.

Tree). Then, we use this mirror to extract information from a remote contract instance deployed in the blockchain (which is encoded as a binary memory layout). The contract data we gathered is exposed in four different formats: (i) data proxy object (REST), (ii) Pharo widget user interface, (iii) JSON, and (iv) HTML.

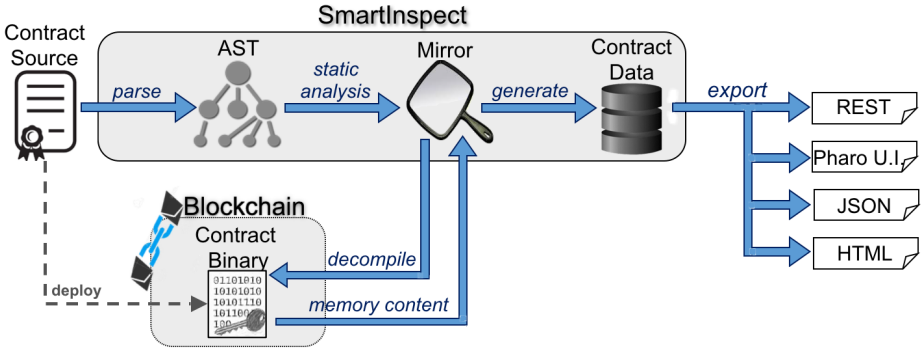

Figure 2: SmartInspect building process.

This approach allows us to access remote structureless information in a structured way. Our solution meets most of the desirable properties that are important for remote de-

bugging namely: interactiveness and distribution [16].

4.2 Discovering the Memory Layout

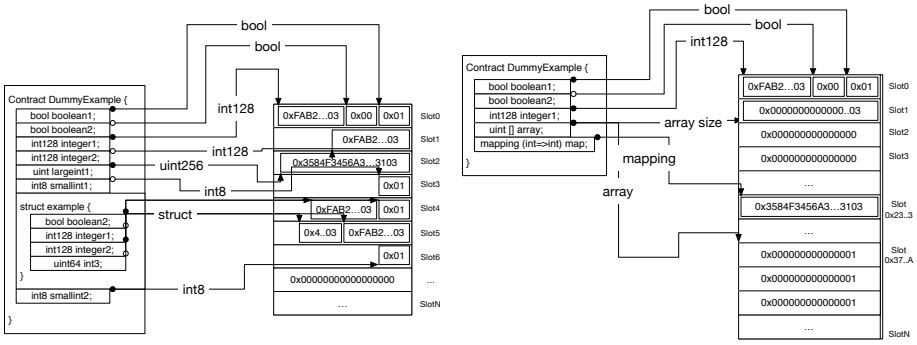

Figure 3: Memory layout representation: (a) Static, (b) Dynamic

First, we needed to decompile the binary representation of a deployed contract (i.e., a contract ABI) to discover the information inside the contract instance. That would resolve the opaqueness problem we described earlier, since we would be able to understand contract attributes.

The Ethereum API provides only one way to access memory layout of a contract: getStorageAt calls. This call gives access to a tree where information is encoded into slots accessible through contiguous indexes, for statically allocated memory (static types), and accessible by Keccack hash for dynamically allocated memory (variable sized arrays and mappings) [8]. It was a big challenge to decompile the memory layout because the Solidity documentation is incomplete. We had to reverse engineer some of the encoding performed by the compiler by ourselves.

There are two key restrictions in memory access: types and order. Each memory slot stores up to 32 bytes. As general policy, the compiler tries to pack as much data as possible for basic types. For example, two booleans and one *int128* occupy 18 bytes from a slot, one byte for each boolean plus 16 bytes for the *int128*. If we add another type that can not fit into the current slot, then the compiler place it in a new memory slot.

In the case of structs, they always start in a new slot and their data may take as many slots as needed. Any data after the struct will be encoded to start in a new slot, no matter if it could fit into the remaining struct slot. We show some of these encoding details for static types in Figure 3a. Then, transversally, all the static representation of the different variables is available contiguously in the first slots of the binary representation, from 0 to *N*.

Dynamically allocated data types (e.g., arrays, mappings) are encoded in other hashed addresses, as shown in Figure 3b[7]. Therefore, the decompilation has to dive into a sometimes contiguous, sometimes indexed slots, with arbitrary allocations of space that may depend on the size of the type or in the next and/or previous type, to be able to read the stored content.

Therefore, we were successful in addressing our concern to introspect a deployed contract data, as we resolved the contract opaqueness problem by decompiling the binary representation and decoding the memory layout.

4.3 Building the Mirror

After we decoded the memory layout, we still needed to apply our solution to any contract for a general reusable solution. We employ a mirror-based architecture [2] that mimics the structure of any contract for us to access the memory layout that we can decode. A mirror works like an independent meta-programming layer which splits the concern of reflection capabilities into a mirror object.

First, we require the contract source code as input to start building the mirror. Then, we parse the source to create an AST (Abstract Syntax Tree). By interpreting the AST, we are able to know every type declared in the contract in the correct order. As described earlier (Section 4.2), we need know the types and order to decoded the memory layout and access the contract data.

Aiming at a general solution, we model configurable mirror objects that allow us to interact with deployed contract instances of the same configuration (usually meaning the same contract deployed in the blockchain).

Our approach builds a composite mirror object, called ContractMirror, whose each component knows how to decode, in order, the contract state (Figure 4, the left side of the diagram). For each variable or struct a corresponding elementary mirror is added to the composite.

At this point we have a mirror with the structural representation of the AST that knows, in order, each contract property with its related type. Now our approach builds a representation of the memory layout to access the stored data. By using static code analysis provided by the AST, we can find the exact place of storage of every contract property. Moreover, we map the contract properties to its corresponding memory slot. Therefore, the mirror uses this mapping to gather the contract data for inspection. Figure 4 (on the right part) shows the mirror's memory mapping as a class diagram.

4.4 Inspection Example

Getting back at our problem example (Section 3.2), where the person in charge (Bob) needed to verify the data in a deployed poll contract instance. Back then, Bob had

[7] We decided to represent the slot as contiguous data to facilitate its representation.

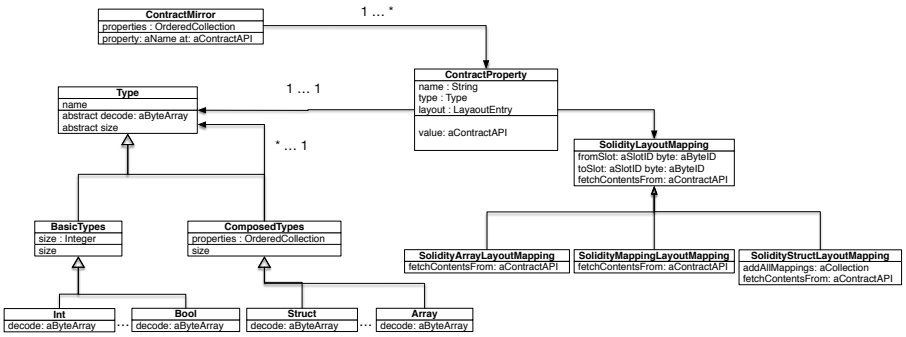

Figure 4: Contract Mirror UML Diagram

only two possibilities to address an user's issue, and both were not feasible. Now, let's suppose that Bob just learned about SmartInspect, which gives him a third possibility, inspect the contract data using our tool. Since Bob is the person in charge he has access to both the contract source code and its deployed binary representation. Bob executes SmartInspect on its poll contract and he is presented with the data from the pollTable property (Figure 5). Finally, Bob can see that the user was not registered, and can now can easily fix the problem by executing the addVoter function on his client application. This simple example illustrates the importance of contract inspection.

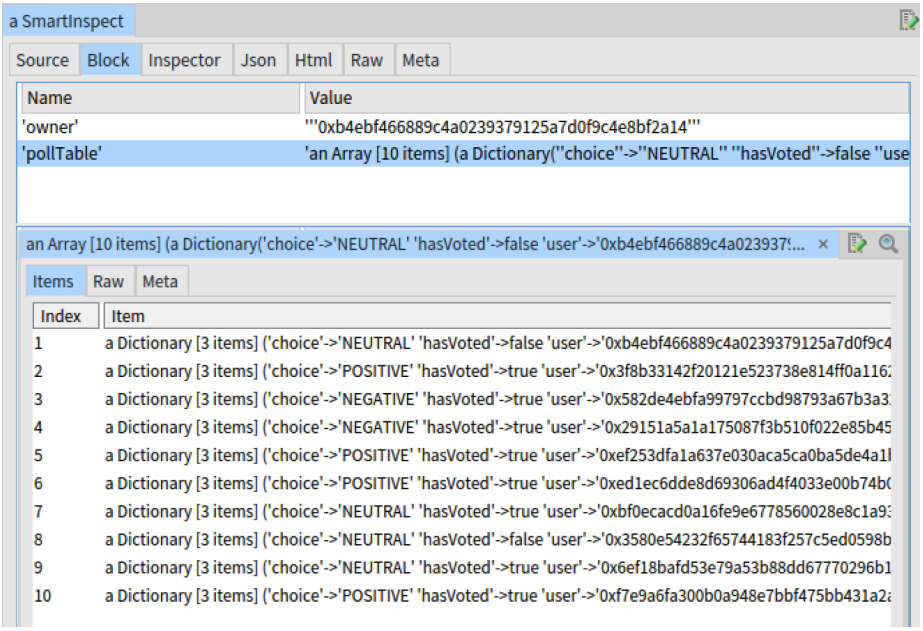

Figure 5: SmartInspect Pharo User Interface Screenshot

5 Preliminary Evaluation

In this section, we present our preliminary evaluation of SmartInspect. The goal of our evaluation was to investigate whether or not SmartInspect implements the necessary and desirable features for an inspector. We used the following four characteristics used for remote debugging by Papoulias [16]: Interactiveness, Distribution, Security, and Instrumentation. We also analyzed other five characteristics that are important for a blockchain remote inspector: Privacy, Pluggability, Consistency, Reusability, and Unrestricted types. We detail the nine characteristics as follows:

- *Interactiveness:* the inspector shows the object's state in real time. A lack of interactiveness could be a problem in blockchain platforms because of the contract's state may change during inspection and the user would be presented with outdated information.
- *Distribution:* the inspector can be extended for other technologies. Ideally, a debugger or inspector should rely on a middleware that is extensible. For smart contracts, the inspector should be extensible over different smart contract languages and blockchain technologies.
- *Security:* since remote debugging access a target through a network, it is important to ensure security from both ends. On the target side, the inspector should not have unrestricted access to its device; this is already ensured by the blockchain platform.
- *Instrumentation:* the inspector can alter the semantics of a process to assist in debugging. Basically, this is the mechanism to halt the process and inspect it at that point (e.g., breakpoints and watchpoints). This characteristic is not possible in blockchain platforms, as we cannot modify the deployed contract code to halt a function in the middle of its execution in the blockchain.
- *Privacy:* inspection should not breach or compromise data privacy by exposing data to unauthorized users. When considering smart contracts, a lack of privacy is dangerous as it could be exploited by malicious users to acquire illicit advantages and resources.
- *Pluggability:* the inspector can be used on existing objects without the need to re-instantiate the objects or the system. For contracts, this means we can inspect existing deployed contracts without any dependency on the contract side, or the need to redeploy the contract. An unpluggable approach has the disadvantage of requiring the redeployment of a contract, which has non-trivial transactional costs.
- *Consistency:* the representation used by the inspector must reveal the information in a consistent manner, i.e., the inspection must reflect the current state of the deployed contract.
- *Reusability:* the inspector can be reused for different contracts. Lack of reusability would require a developer to spend time redefining the inspection for each

individual contract.

- *Unrestricted Types*: the inspection can handle all types of objects. In contrast, a type-restricted inspector supports only a subset of data types (e.g., primitive types, static types).

We analyzed SmartInspect according to the characteristics of inspection tools just presented. Even though we wanted to compare SmartInspect against related approaches, as far as we know, there are no other inspectors available for Solidity smart contract. Therefore, we compared our approach against two other practices to access contract data: Getter methods, and ad-hoc Decoder (Table 1).

Table 1: Related techniques comparison

Characteristic	**SmartInspect**	**Getter**	**Decoder**
Interactiveness	Yes	Partial	Partial
Distribution	Yes	No	No
Security	Yes	Yes	Yes
Instrumentation	No	No	No
Privacy	Yes	No	Yes
Pluggability	Yes	No	Yes
Consistency	Yes	Yes	Yes
Reusability	Yes	No	No
Unrestricted Types	Yes	No	Yes

As we can see from Table 1, SmartInspect's only characteristic flaw is related to *instrumentation* for remote debugging. However, this is a blockchain limitation rather than a problem in our tool.

Getter methods are a simple solution, since they are cheap to implement and easy to test. The developer does not need to know the memory layout of a contract to create getter methods. However, the developer forgets to make a getter for a given attribute, he/she will need to re-deploy the contract and, most often, lose the data from the previous instance. Solidity does not support the return of many complex types (e.g, structs, mappings) on its functions. Therefore, a developer might need to adapt his/hers data or function to provide access to a complex type. Moreover, the easy access to the data may cause a loss of privacy, since getter methods are a public part of the contract binary enconding.

Another practice is the ad-hoc Decoder that uses the Ethereum API on the memory slots. This is a complex task since it demands a deep understanding of the memory layout of each contract a developer plans to inspect. It also requires a developer to know the type of each attribute and code the ad-hoc decoder accordingly. Its advantages are that it allows access to data without loss of privacy and without the need to redeploy the contract. In fact, SmartInspect uses this concept of decoding memory layouts as a part of its inspection process.

6 Discussion

In this section, we discuss our evaluation (Section 6.1) and the possible benefits for inspecting smart contracts (Section 6.2).

6.1 Evaluating the Inspector

In our preliminary evaluation, we compared SmartInspect against two practices that can be used to access contract data (Section 5). We acknowledge that we need other inspectors for a better comparison, since practices are a less polished solution than a fully designed approach for inspection. However, as far as we know, there is no other inspector tool available for Ethereum blockchain.

In the future, we plan to improve the evaluation by comparing with other inspectors, and performing a more extensive user driven evaluation.

6.2 Benefits from Inspecting a Contract

There are several benefits for inspecting a smart contract on a blockchain platform. We highlight the following:

- **Easier to Understand.** Blockchain provides a mechanism that can be used to build trust between entities without a middleman [7, 15]. In the blockchain environment, smart contracts became a popular way to transfer digital assets among such entities [1, 13]. From a beginners perspective, it is better to work with concrete examples to understand a new concept. Thus, an inspector provides a simple way to access the contract state, which facilitates its understanding.
- **Find Bugs.** Contract inspection can help developers to find bugs more easily (as we illustrated in Section 3.2).
- **Transparency.** Supporting inspection of contracts can increase transparency and improve overall trust among entities dealing with blockchain. For instance, it is possible for two entities to show the current state of their contracts to each other promoting transparency in their interactions.
- **Encourage the adoption of Contracts**. By allowing contract inspection, we can promote trust and transparency on blockchain platforms to companies and institutions. This will encourage more people to adopt smart contracts for their business or academic activities.

7 Related Work

We organized the related work into three groups: (i) inspecting and debugging, (ii) reverse engineering, and (iii) blockchains and smart contracts.

7.1 Inspecting and Debugging

Chis et al. [4] performed an exploratory research to better understand what developers expect from object inspectors, and based on that feedback they propose a novel inspector model. The authors interviewed 16 developers for a qualitative study, and a quantitative study conducting an online survey where 62 people responded. Both studies were used to identify four requirements needed in inspector. Then they propose the Moldable Inspector, which indicates a new model to address multiple types of inspection needs. We followed the lessons taken by the Moldable Inspector when creating SmartInspect. We deem noteworthy the multiple views aspect, as SmartInspect can present its inspected data in three different views.

Papoulias [16] gives a deep analysis on remote debugging. As discussed by the author, remote debugging is specially important for devices that cannot support local development tools. The author identifies four important characteristics for remote debugging: interactiveness, instrumentation, distribution, and security. Based on the identified properties, Papoulias proposed a remote debugging model, called Mercury. Mercury employs a mirror based approach and an adaptable middleware. We used Papoulias research as an inspiration to create SmartInspect, specially relying on mirror for the remote inspection.

Salvaneschi and Mezini [17] propose a methodology, called RP Debugging, to debug reactive programs more effectively. The authors discuss that reactive programming is more customizable and easier to understand than its alternative the observer design pattern. The authors also present the main problems and challenges to debug reactive programs, and the main design decisions when creating their methodology. Although our inspector is from a different application domain, the RP Debugging design served as inspiration to plan our own inspecting approach.

7.2 Reverse Engineering

Srinivasan and Reps [18] developed a reverse engineering tool to recover class hierarchical information from program binary. Their tool also extracts composition relationships as well. They use dynamic analysis to obtain object traces and then they identify the inheritance and composition information among classes on those traces. The authors experiments show that their recovered information is accurate according to their metrics. The author's tool contrasts with SmartInspect as, we use static analysis and they use dynamic analysis.

Caballero et al. [3] propose an approach to reverse engineer protocols by using dynamic analysis on program binaries. As stated by the authors, this approach differs from others that extract protocol information purely from network traces. The authors argue the importance to extract the protocol information, when there is no access to its specification, for network security applications. They used 11 programs that implemented five different protocols to evaluate their approach.

Fisher et al. [9] propose a multi phase algorithm to process *ad hoc data* without human interaction. The authors describe *ad hoc data* as semistructured information that does not have tools easily available. Basically, their algorithm reverse engineer the *ad hoc data* into a domain specific language, which is used to generated a set of tools such as parsers, printers, query engine, and others. The authors evaluate the performance and correctness of their approach by using different benchmarks.

Lim et al. [12] designed an analysis tool called File Format Extractor for x86 (FFE/x86). They extract information from an executable file to perform several process, including static analysis. Their evaluation consists in applying their tool in three systems: gzip, png2ico, and ping. The authors work is similar to SmartInspect in a way that both approaches use static analysis in one of its steps to enhance the reverse engineering.

7.3 Blockchain and Smart Contracts

Dinh et al. [5] describe a benchmarking framework to analyze private blockchain platforms. The authors contrast the different among public blockchain platforms (e.g., Ethereum) and private ones. For instance, private blockchain show more focus towards secure authentication. Although, their framework was designed for private blockchains, they evaluate it using public blockchain as well. Their evaluation measured four aspects (throughput, latency, scalability, and fault tolerance) in three blockchains: Ethereum, Parity, and Hyperledger. The benchmark framework provided an in dept analysis of blockchain platforms that we used when we designed SmartInspect.

Luu et al. [13] investigates the security problems of executing smart contracts on the Ethereum platform. They also propose solutions to make the contracts more secure. The authors presents several scenarios and the possible malicious exploits for those scenarios. Based on the presented vulnerabilities, they propose solutions to make contracts more secure. The authors also propose a tool, called Oyente, that flags potential security flaws when coding smart contracts. Similarly to SmartInspect, Oyente also uses the contract bytecode to makes its security recommendations. However, our tool uses the memory layout to access the data, and Oyente uses the bytecode for a symbolic execution and security analysis.

Bhargavan et al. [1] proposed a framework to convert contracts to F* and then improve their security. F* is a functional programming language proposed by the authors in their work. According to them, F* was designed to better verify correctness of contracts. Their approach decompiles the contract bytecode into a special F* code to verify low-level properties; similarly it also compiles a contract source into an F* version to verify high-level properties. The authors did a preliminary evaluation where 46 contracts were

translated to F*. Their approach also employs decompilation of contract bytecode and parses source code, similar to our SmartInspect.

8 Conclusion

In this technical report, we present the specific problems for inspecting Solidity smart contracts. Smart contract opaqueness added to the problems of reverse engineering compiler encoding and packing on different types of entity makes the inspecting of values in smart contracts almost impossible for regular developers. Our approach implementation, SmartInspect, is a local pluggable mirror-based reflection system for remotely deployed objects on reflection-less systems (the Ethereum platform). SmartInspect allows the inspection of known contracts based on it source code focusing on the debugging properties of interactiveness and distribution. This reflective approach allows a user to see the contents of any contract instance of the given source code, without needing to redeploy, nor develop any ad-hoc code.

We planned the following ideas for future work: (i) extend SmartInspect to inspect other smart contract languages employed in the Ethereum platform besides Solidity; (ii) extend SmartInspect to support other blockchain platfroms; (iii) improve the introspection capabilities to support full debugging on smart contracts; and (iv) improve the evaluation by using SmartInspect on a big contract database, comparing it with other inspectors, and performing a more extensive user evaluation.

Acknowledgment

We like to thank the UTOCAT company for giving valuable feedback to our research.

References

[1] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

[4] A. Chiş, O. Nierstrasz, A. Syrel, and T. Gîrba. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[5] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1085–1100, New York, NY, USA, 2017. ACM.

[6] S. Dziembowski. Introduction to cryptocurrencies. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1700–1701, New York, NY, USA, 2015. ACM.

[7] Ethereum Foundation. Ethereum's white paper., 2014.

[8] Ethereum Foundation. Solidity documentation release 0.4.18. Technical report, 2017.

[9] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 421–434, New York, NY, USA, 2008. ACM.

[10] A. Hari and T. V. Lakshman. The internet blockchain: A distributed, tamper-resistant transaction framework for the internet. In *15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 204–210, New York, NY, USA, 2016. ACM.

[11] B. Leiding, P. Memarmoshrefi, and D. Hogrefe. Self-managed and blockchain-based vehicular ad-hoc networks. In *2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, pages 137–140, New York, NY, USA, 2016. ACM.

[12] J. Lim, T. W. Reps, and B. Liblit. Extracting output formats from executables. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 167–178, 2006.

[13] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 16)*, pages 254–269, New York, NY, USA, 2016. ACM.

[14] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[15] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system., 2009.

[16] N. Papoulias. *Remote Debugging and Reflection in Resource Constrained Devices*. Theses, Université des Sciences et Technologie de Lille - Lille I, Dec. 2013.

[17] G. Salvaneschi and M. Mezini. Debugging reactive programming with reactive inspector. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 728–730, New York, NY, USA, 2016. ACM.

[18] V. Srinivasan and T. Reps. *Recovery of Class Hierarchies and Composition Relationships from Machine Code*, pages 61–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.