SmartConDetect: Highly Accurate Smart Contract Code Vulnerability Detection Mechanism using BERT

Sowon Jeon, Gilhee Lee, Hyoungshick Kim, Simon S. Woo
College of Computing and Informatics / Department of Electrical and Computer Engineering
Sungkyunkwan University, South Korea
{soonej40,2gilhee,hyoung,swoo}@skku.edu

ABSTRACT
Many popular blockchain platforms support smart contracts, which are the programs executed, and stored as transactions on their blockchain protocols and execution environments. However, it is not easy to develop secure smart contracts since smart contracts are programs that can often have security vulnerabilities, which may lead to severe financial loss to service providers or users. Therefore, it is critical to detect security vulnerabilities in smart contracts. In this paper, we propose SmartConDetect to detect security vulnerabilities in smart contracts written in Solidity. SmartConDetect is designed as a static analysis tool to extract code fragments from smart contracts in Solidity and further detect vulnerable code patterns using a pre-trained BERT model. To show the feasibility of SmartConDetect, we evaluate the performance of our approach with 10,000 real-world smart contracts collected from the Ethereum blockchain platform. Our experimental results demonstrate that SmartConDetect outperforms all state-of-the-art methods.

CCS CONCEPTS
• Security and Privacy → Software security engineering; Software reverse engineering; Vulnerability management.

KEYWORDS
BERT, Pretrained model, NLP, Solidity, Smart Contract, Ethereum, Blockchain

ACM Reference Format:

1 INTRODUCTION
Source code analysis is a technique to examine source code before a program is run, allowing us to detect vulnerabilities and flaws in a program. Especially, analyzing and detecting vulnerability in source code for financial applications in a blockchain is critical, as it can cause severe economic problems. So far, many code analysis techniques have been proposed. One of the popular approaches is to convert program source code into its intermediate representations using Abstract Syntax Tree (AST) [13], and the Data Flow Graph (DFG) [4]. Recently, several natural language processing (NLP) techniques have been applied to process programming languages such as code generation and representation, optimization, and synthesis [1, 20, 24, 26] as well as detecting bugs [19, 25].

Smart contracts [11] are the program stored on a blockchain for the verification, control, or execution of an agreement, which is a fundamental feature supported by blockchain platforms such as Ethereum. All transactions are executed automatically without a third party through smart contracts. Therefore, we can develop decentralized apps (dApps) based on these characteristics of smart contracts. Recently, many dApps (e.g., financial services, games, etc.) have been introduced in online marketplaces. Smart contracts on Ethereum cannot be modified or deleted due to integrity once deployed. Therefore, developers have to ensure no vulnerabilities before deploying smart contracts. However, it is challenging for developers to detect vulnerabilities in smart contracts. Previous studies showed that many real-world smart contracts deployed on Ethereum have severe vulnerabilities [3] that can be abused by attackers to damage the blockchain systems seriously. For example, there was a severe incident called the “The Decentralized autonomous organization (DAO) attack” [18], which was to exploit the recursive call vulnerability to transfer one-third of the DAOs funds to a malicious account (the affected Ether had a value of about $50M). Therefore, many researchers have performed the analysis of smart contract vulnerabilities [2, 12, 15]. However, the state-of-the-art NLP models have rarely been considered for detecting vulnerabilities of smart contracts. To the best of our knowledge, there has been no research that applied Bidirectional Encoder Representations from Transformers (BERT) [6] to detect the vulnerability, on Solidity [9] which is an object-oriented programming language for writing smart contracts on Ethereum.

Therefore, in this work, we propose a highly accurate vulnerability detection mechanism of smart contracts using BERT, SmartConDetect, for improving the security of smart contracts. SmartConDetect can be deployed in the real-world scenario in two different ways. First, developers can use SmartConDetect to validate their smart contract codes before defining them as transactions. Through this, the developer can reduce the security risk resulted from the source code. Secondly, blockchain service providers such as Ethereum can provide a secure transaction environment with our SmartConDetect, where SmartConDetect can validate the Solidity code before permitting the execution of the smart contract. For example, if SmartConDetect discovers a severe vulnerability, the platform can deny executing the smart contract. Furthermore,
SmartConDetect does not require an additional compiling process because SmartConDetect can find vulnerabilities from the source code directly. Our contributions are summarized as follows:

- We propose SmartConDetect, the BERT-based smart contract vulnerability detection model, which extracts code gadgets, leveraging BERT to detect the vulnerability of Solidity programs effectively.
- We evaluate our model with our own smart contract dataset, including 10,000 Solidity files and vulnerability labels.
- We demonstrate that SmartConDetect outperforms the state-of-the-art methods, achieving a higher F1-score, comparing to other baselines, including SVM, Eth2Vev, and GNN.

2 RELATED WORK

BERT. Bidirectional Encoder Representations from Transformers (BERT) [6] consists of a multi-layered Transformer encoder, and has been highly successful in many natural language processing areas. In particular, BERT has been applied to semantic analysis, translation, and question and answering. Furthermore, many researchers have used BERT for analyzing source code [5, 8, 14]. CodeBERT [8] is a bimodal pre-trained model for programming language (PL) and natural language (NL), which is a transformer-based model. CodeBERT is applied to natural language code search and code documentation generation. CuBERT [14]. Code understanding BERT, shows the efficacy of the pre-trained contextual embedding in Python on five classification tasks: variable-misuse classification, wrong binary operator, swapped operand, function-docstring mismatch, exception type, and variable-misuse localization and repair. However, we focus on detecting security vulnerabilities in the smart contract written in Solidity.

Smart Contract. A smart contract [21] is a script that automatically executes a contract online. Until the advent of Ethereum, the smart contract has existed primarily as a concept. However, smart contracts have been increasingly grown with Ethereum, and many dApps using smart contracts are currently being actively developed. When developers create and distribute smart contracts, users can use the functions through the smart contract or automatically execute them according to the contract conditions. Blockchain stores the data in a chain format, so it is difficult to modify or delete data once stored. Therefore, developers must check for errors or vulnerabilities before deploying the smart contracts. However, it is challenging for developers to prevent vulnerabilities, and for this reason, many smart contracts with vulnerabilities are deployed on the Ethereum network. Thus, detecting vulnerabilities in smart contracts is essential.

Vulnerability Detection in Smart Contracts. The analysis of vulnerabilities in the smart contract has been studied in several works. SmartCheck [23] is a Solidity static analysis tool that is composed of 43 pre-categorized vulnerability rules. SmartCheck extracts patterns by converting the source code to XML, but it has limitations in finding sophisticated vulnerabilities or new types of vulnerabilities. Zhuang et al. [12] have proposed a Degree-Free Graph Convolutional Neural network (DR-GCN) and a Temporal Message Propagation network (TMP), which learn from the normalized graphs for detecting vulnerability. They construct the graph structures using the importance of program elements in functions.

Momeni et al. [16] utilized machine learning models for analyzing the smart contract vulnerabilities, where they used Support Vector Machine, Neural Networks, Random Forest, and Decision Tree for evaluation, achieving 95% accuracy on average. Eth2vec [2] is a machine learning-based static analysis tool for finding vulnerabilities and is particularly strong against code rewriting attacks. Eth2vec uses Ethereum Virtual Machine bytecodes (EVM bytecodes) rather than Solidity source code. However, it requires manually created vulnerability features to ensure its detection accuracy. On the other hand, ESCORT [15], Deep Neural Network (DNN) based framework for smart contract vulnerability detection, is proposed to detect multiple vulnerabilities using transfer learning. However, to the best of our knowledge, no prior research has leveraged the BERT to detect vulnerability from Solidity-based smart contracts. Also, multi-class classification is not dealt with so far. SmartConDetect is capable of detecting 23 vulnerabilities with extensive vulnerability information.

3 METHODOLOGY

SmartConDetect is a BERT-based smart contract vulnerability detector, where our model consists of the training and detection phase as shown in Fig. 1. Essentially, SmartConDetect is a code-similarity-based approach, which computes the similarity between code fragments via their abstract representations from the tokenizer process.

3.1 Training

Preprocessing. The input to the training phase is a large number of Solidity codes, where some of which are vulnerable and the others are not. We use 23 different vulnerability classes, leveraging the outputs from Smartcheck as the ground truth. In Appendix A, we explain the vulnerability classes and the example codes. For example, Fig. 2 shows Solidity code with the vulnerability named ‘visibility’ (Vis). In line 9 and 10, the createNewAmount function is declared without visibility (external, internal, public, and private), hence it is classified as a vulnerability. Because visibility enables access control, it prevents unauthorized or malicious users from executing sensitive functions. The sensitive functions should not be accessible to anyone, such as createNewAmount that issues new tokens. As the output of the training phase, we classify the code vulnerability patterns. In Figure 1, we present the overall architecture of the training phase. We first extract functions from the Solidity codes, and label each function using the ground truth labels of vulnerabilities detected from Smartcheck. Then, we generate code gadgets text files by assembling extracted functions. We do not consider the inheritance and semantic relations, as we do not have vulnerabilities to detect between functions.

Training. We first prepend a [CLS] token to indicate the beginning of the functions and append a [SEP] token to represent the end of the code line. Then, after masking, we replace the 10% tokens with random tokens, using our vocab list that includes the Solidity expressions. In particular, we collected this vocab list from the official language description released by Ethereum [9].

We train SmartConDetect using BERT. Having encoded the code gadgets into vectors and obtained their ground truth labels, we follow the standard training process for learning BERT, as shown in Fig. 1. In the BERT, we process the output feature vector into a
SmartConDetect: Highly Accurate Smart Contract Code Vulnerability Detection Mechanism using BERT

Figure 1: Overall architecture of SmartConDetect, where SmartConDetect has two phases: (1) Training Phase and (2) Detection Phase. First, we extract code gadgets from Solidity files and tokenize codes with [CLS] and [SEP] tokens, where we use these tokens as an input of BERT. We train BERT to learn the representation of vulnerabilities by using these inputs. Finally, we classify the vulnerabilities and present JSON-type reports with filename, vulnerability rules, and corresponding functions.

Figure 2: Example of smart contract vulnerability (Visibility).

fully connected layer to reduce the embedding output. Finally, the softmax function is used as an activation function and the binary cross-entropy function at the classifier.

3.2 Detection
To detect vulnerability from Solidity codes, we extract the functions from the code and the code gadgets of the training phase. We provide the code gadgets after tokenization as an input to SmartConDetect, as shown in Fig. 1, where the network outputs are the vulnerability classes of each function. SmartConDetect automatically generates the output file as a detection report into JSON type, containing the testing file names, the vulnerability rule name, and the corresponding vulnerable functions, as shown in Fig. 1. Through this process, the developers can detect the vulnerabilities of the smart contracts before deploying them.

4 EXPERIMENTAL RESULTS

4.1 Experimental settings
Dataset Description. We use 10,000 Solidity files collected using Etherscan [22]. We have collected our dataset during a period of four weeks, from April 2021 to May 2021. Since we do not use any compilers for preprocessing, we do not consider the compiler version of the source code. For evaluation, we split the dataset into 80% training, 10% as a validation set, and 10% testing set.

Setup. Our experiment setting is as follows: we use Intel(R) Xeon(R) Silver 4114 2.20 GHz CPU with 256.0 GB RAM and NVIDIA GeForce Titan RTX for building our model. We use Pytorch for implementing our code. Next, we start our model with the learning rate of 0.00001 and train the model with three epochs. We set the random seed to 2018 and batch size to 24. For the BERT_base model, we use the one with 12 layers of transformer blocks, 768 hidden layers, and 12 self-attention heads. We use the Adam optimizer and cross-entropy for the classification loss.

Baselines. We use the following machine learning-based (SVM [16]), unsupervised learning-based (Eth2Vec [2]) and graph neural network-based (DR-GCN [12]) methods as our baselines:

- **SVM** [16]: Momemi et al.’s model [16] uses abstract syntax tree (AST) as inputs. We extract AST and implement SVM by using Scikit-learn [17].
- **Eth2Vec** [2]: Eth2Vec uses EVM bytecodes as input. Thus, we compile our dataset using py-solc [10]. We implement Eth2Vec by using Kam1n0-server version 2.0.0 [7] to follow the experiment setting of Eth2Vec.
Figure 3: AUC of each vulnerability rules. (Blue: AUC under 90%, Orange: AUC higher than 90% and lower than 100%, Green: AUC 100%).

Table 1: Vulnerability Detection Performance of SmartConDetect.

<table>
<thead>
<tr>
<th>Models</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>F1-Score (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM [16]</td>
<td>59.3</td>
<td>42.7</td>
<td>45.2</td>
</tr>
<tr>
<td>Eth2Vec [2]</td>
<td>77.0</td>
<td>50.7</td>
<td>57.5</td>
</tr>
<tr>
<td>DR-GCN [12] (Ours)</td>
<td>74.0</td>
<td>82.6</td>
<td>78.1</td>
</tr>
<tr>
<td>SmartConDetect (Ours)</td>
<td>98.7</td>
<td>86.2</td>
<td>90.9</td>
</tr>
</tbody>
</table>

- DR-GCN [12]: We use the default configuration of DR-GCN. And we set learning rate as 0.001, dropout rate as 0.1 in 50 epochs.

Evaluation metrics. We use the widely used precision, recall, and F1-score metric to evaluate SmartConDetect. More specifically, we prefer to achieve a low false negative rate (FNR) and a false positive rate (FPR), which means higher precision and recall. We also compare the AUC of each vulnerability rule to analyze the performance.

4.2 Results
In Table 1 and Fig. 3, we present our experimental result. As shown in the Table 1, SmartConDetect outperforms baseline models. Examining the F1-score results, BERT achieves 90.9%, F1-score. Compared to other vulnerability analysis tools, our system shows a significant performance improvement. The precision is improved up to 39.4% compared to the worst-performing SVM and 24.7% compared to the best performing GNN. Also, SmartConDetect shows better results than the baselines in the recall.

Figure 3 shows the AUC of each vulnerability rules when using BERT. We can find that most of the rules achieve high AUC, higher than 90%, except for four rules (MEZ, ALM, var, AMERF). Furthermore, this result implies that our SmartConDetect can detect the various vulnerabilities. However, for those rules which are not showing high AUC, we surmise that BERT may be overfitted to the vulnerability rules used, which are pretty short. We assume that the model learns the feature of vulnerability syntax with noise if the syntax is too short. In summary, SmartConDetect is a code-similarity-based approach, which computes the similarity between code fragments via their abstract representations from the tokenizer process. Thus, it has a limitation on detecting certain types of vulnerabilities with short Solidity syntax.

5 CONCLUSION
In this work, we propose a novel BERT-based vulnerability detection model of smart contract, SmartConDetect, and demonstrate that BERT achieves high accuracy on detecting vulnerabilities compared to existing popular baseline models. We also have generated a large-scale dataset that includes Solidity source code and vulnerability labels. Although we trained our model with only Solidity source code, SmartConDetect can be applied to any language by providing the vocab list of specific programming languages such as Go and Python into the training process of BERT. As a part of future work, we plan to develop a more advanced and improved model to detect recently discovered novel security vulnerability types using few-shot learning.

ACKNOWLEDGMENTS
This work was partly supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-00421, AI Graduate School Support Program (Sungkyunkwan University)), (No. 2019-0-01343, Regional strategic industry convergence security core talent training business) and the Basic Science Research Program through National Research Foundation of Korea (NRF) grant funded by Korea government MSIT (No. 2020R1C1C1006004). Also, this research was partly supported by IITP grant funded by the Korea government MSIT (No. 2021-0-00017, Original Technology Development of Artificial Intelligence Industry), and was partly supported by the MSIT (Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program (2020-0-01550) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation).
REFERENCES


A APPENDIX

In Appendix, we explain about 23 vulnerability rules we detect in SmartConDetect. The vulnerability rules are first defined by [23], and we have reorganized the rules by combining related rules.

A.1 Smart Contract Vulnerability Rules

A.1.1 Gas exhaustion (GE). Blockchain requires gas to execute transactions and stop working when the gas runs out. When the length of the array in the smart contract is long enough, it may exceed the amount of allocated gas. If anyone can add data to the array, the attacker can create the DoS attack by increasing the size of the array.

Listing 1: UFC Example

```
call (abi.encodeWithSelector(SELECTOR, to, value))
```

A.1.2 Unchecked function call (UFC). When using a function that calls a function of another contract (call, delegatecall, send), the developer must take measures against failure. Therefore, the developer must check the result of the function call. In particular, send that transfers ether must be checked, and it is recommended to use transfer function instead.

Listing 2: UFC Example

```
call (abi.encodeWithSelector(SELECTOR, to, value))
```

A.1.3 Function that should not be view (FSNV). Among the functions of Solidity, the view function is a function that consumes less gas. So, it cannot use selfdestruct or externally calling function, assembly inside the function.

Listing 3: FSNV Example

```
function isContract(address addr)
    internal view returns (bool) [uint256 size;]
    assembly {size := extcode_size (addr)}
    return size > 0;
```

A.1.4 Function that should not be pure (FSNP). There are several types of functions (view, pure, constant, payable) in Solidity, and a pure function should use less or no gas consumption than a view function. Thus, some features are not available in pure functions.
- Unable to read data at the storage level.
- Cannot call event.
- Cannot create other contracts.
- Cannot use selfdestruct function to destroy current contract.
- Cannot use functions such as ether transfer.
- Cannot call the external functions that are the type of view and pure.
• Unable to run inline assembly.
• Unable to check balance of the contract (cannot use this.balance).
• Cannot use msg and tx to get the information about Ethereum.

Listing 4: FSNP Example
1 function getChainId()
2 internal pure returns (uint)[ uint256
3 chainId: assembly{chainId:=chainid( )}
4 return chainId; ]

A.1.5 Should not check this.balance equality directly (SNCBED). Solidity supports this.balance to check the balance of the smart contract. Be careful with using == or != when checking the balance with this.balance. Comparing this.balance directly is not recommended because the attacker can maliciously modify the value of it.

Listing 5: SNCBED Example
1 address(this).balance!=0

A.1.6 Functions returns incorrect type or nothing (FRITN). When implementing a function, if a return value is declared but does not return any value or the return value type does not match, it is classified as a vulnerability. Unused return variables allow attackers to exploit, and mismatching variable types can lead to errors in other data.

Listing 6: FRITN Example
1 function implementation() internal view
2 returns(address impl){ bytes32 slot=
3 IMPLEMENTATION_SLOT; assembly{impl:=
4 sload(slot)}

A.1.7 msg.value equals zero (MEZ). Users can send ether by putting it in the transaction to use the smart contract and check it through msg.value. Users can send 0 ether. In general, zero-value transactions are not a threat but can be exploited by attackers.

Listing 7: MEZ Example
1 msg.value==0

A.1.8 Visibility (Vis). To control access to all variables and functions, Solidity supports visibility option. Visibility consists of four types: external, internal, public, and private. All variables and functions must be declared visibility for security, and constructors must be declared internal or public.

Listing 8: Vis Example
1 function Ownable() {owner=msg.sender ;}

A.1.9 Array length manipulation (ALM). Users should not modify the array length directly. If the developer need to change it, the developer need to store the length of the array in another variable and then modify it. The function that directly changes the length of the array allows the attacker to resize the array. And loops using that array can run infinitely or run out of gas (It is related to A.1.1).

Listing 9: ALM Example
1 presaleGranteesMapsKeys.length--

A.1.10 SafeMath library (SML). Using normal arithmetic operations (+, -, *, /) can cause the overflow or underflow for integer types. In particular, overflow can be exploited by attackers to steal ether. Therefore, the developer should avoid using arithmetic operations from Solidity’s SafeMath library or using conditional statements.

Listing 10: SML Example
1 using SafeMath for uint256 ;

A.1.11 Redundant fallback reject (RFR). The smart contract should reject unexpected payments. Prior to Solidity version 0.4.0, the automatic fallback function was not supported, so developers had to create their own function to do this. However, from a higher than 0.4.0 version, the fallback function is automatically supported, which is redundant. If the user is rejected to send 1 ether, the fallback function will be used twice and the user will receive 2 ether.

Listing 11: RFR Example
1 function() {
2 revert();
3 }

A.1.12 Locked the ether (LE). It is critical to declare only functions that accept ether when implementing a smart contract. If victims send their ether to the contract, the victims cannot recoup their ether, and their funds are locked.

Listing 12: LE Example
1 contract LnProxyImplisLnAdmin[LnProxyBase
2 public proxy; LnProxyBase public integration Proxy;
3 address public message Sender;
4 constructor(address payable_proxy) internal
5 {require(admin!=<missing '>';' -> address(0),
6 "Admin must be set");
7 proxy=LnProxyBase(_proxy);
8 emit ProxyUpdated(_proxy);}]

A.1.13 private don’t hide data (PDHD). Solidity provides access control as private function. Some of the developers misunderstand that the private option hides the data and declares the sensitive data such as the secret key as private, which must be hidden. However, private option only prevents access to data, not hide it. Using private option can expose sensitive data since the smart contract is publicly readable.
### SmartConDetect: Highly Accurate Smart Contract Code Vulnerability Detection Mechanism using BERT

#### PLP 2021, August 15, 2021, Virtual Conference

---

**Listing 13: PDHD Example**

```solidity
contract OpenWallet {
    struct Wallet {
        bytes32 password;
        uint balance;
    } mapping(uint => Wallet) private wallets;
    function replacePassword(uint _wallet, bytes32 _previous, bytes32 _new) public {
        require(_previous == wallets[_wallet].password);
        wallets[_wallet].password = _new;
    }
    function deposit(uint _wallet) public payable {
        wallets[_wallet].balance += msg.value;
    }
    function withdraw(uint _wallet, bytes32 _password, uint _value) public {
        require(wallets[_wallet].password == _password);
        require(wallets[_wallet].balance >= _value);
        msg.sender.transfer(_value);
    }
}
```

**A.1.14 approve function in ERC20 library (AFE).** ERC20 is the standard API for transferring and managing ether used by Solidity. ERC20 provide approve function that transfers one’s ether to another person. An attacker can exploit this function to obtain ethers. For example, suppose Alice calls approve to transfer N ethers to Bob. After that, Alice calls approve again to modify the amount of ethers into M. Bob notices this and takes N ethers from Alice before being modified to M. Then, after approve is recorded in the block, Bob can extort M ethers. Alice intend to transfer only M ethers to Bob, but Bob can take N+M ethers by attack.

**Listing 14: AFE Example**

```solidity
function approve(address_spender, uint_value) returns(bool success){allowed[msg.sender][_spender]=_value;
Approval(msg.sender,_spender,_value);
return true;}
```

**A.1.15 Solidity compiler upgrade to 0.5.0 version (050V).** While upgrading the Solidity compiler version to 0.5.0, new rules are added. However, some developers are unaware of the changes, and this can lead to vulnerabilities. Updated rules are as follows:

- When using an array, structure, or mapping, the developer must select the space to store data: storage (a space where state variables are stored permanently), memory (temporary storage space within functions), and calldata (storage space used by external functions).
- When casting, the byte size must be the same.
- When using the call function, use abi.encodeWithSignature instead of using only 4 bytes of the hash value of the function name to be called.

**Listing 15: 050V Example**

```solidity
bytes call data
```

**A.1.16 var (var).** var automatically selects the variable type at the compile time. For example, the 256-bit variable can be declared to be an 8-bit variable. Thus, it can cause overflow problems. From the 0.5.0 version, var variables are not available, but the developers can still use var in the earlier versions.

**Listing 16: Var Example**

```solidity
var premine=Preminer(recipient, monthlyPayment, 0, false, 0)
```

**A.1.17 Function that have return value should not be internal and private (FRVNIP).** Multiple return values should be in the type of struct, instead of several values of internal or private functions. It can improve code readability.

**Listing 17: FRVNIP Example**

```solidity
function test() internal returns(uint a, address b, bool c) {
    a = 1;
    b = msg.sender;
    c = true;
}
```

**A.1.18 transfer function in loop (TFL).** If a transfer that transfers ether to another account is declared in the loop, the attacker can use it to keep sending ether to itself. Therefore, the transfer should always be used outside the loop.

**Listing 18: TFL Example**

```solidity
for(uint i=0; i<preminers[_newPreminer].allocationsCount; i++)
    {preminers[_newPreminer].allocations[i]=oldPreminer.allocations[i];}
```

**A.1.19 Using inline assembly (UIA).** The inline assembly provides developers with low-level access to the Ethereum virtual machine, allowing Solidity programs to access EVM commands and perform tasks directly. Therefore, starting with version 0.5.0, the inline assembly is deprecated.

**Listing 19: UIA Example**

```solidity
assembly{log0(add(_callData,32),size)}
```
A.1.20  **Don't hardcode the address (DHA).** Hardcoding the account’s address or contract’s address in code is not recommended.

**Listing 20: DHA Example**

```solidity
(address(0), delegates[dst], amount)
```

A.1.21  **Deprecated constructions (DC).** Unused constructors should be deleted before deploying the contract.

**Listing 21: DC Example**

```solidity
class BreakThisHash {
    bytes32 hash;
    uint birthday;
    constructor(bytes32 _hash) payable {
        hash = _hash;
        birthday = now;
    }

    function kill(bytes password) external {
        if (sha3(password) != hash) {
            throw;
        }
        suicide(msg.sender);
    }
}
```

A.1.22  **Attacker can make ERC20 functions always return false (AMERF).** In the ERC20 library, functions that transmit or control ether are declared. Functions that transmit ether should return a bool value indicating success or failure. Through this, the developer can check whether the transmission was successful. The attacker makes always returns false, which is not true. Or the attacker can make always fail.

**Listing 22: AMERF Example**

```solidity
function transfer(address dst, uint256 amount) external returns (bool){
    dst; delegateAndReturn();
}
```

A.1.23  **Revert require (RR).** Conditional statements must not depend on the result of external functions. The attacker can manipulate the external function to make conditional statements not consistently successful or produce unexpected results.

**Listing 23: RR Example**

```solidity
if (!assertion) throw;
```