A Review Analysis on Smart Contract Vulnerabilities Using Blockchain

Bibin Baby[1], Alan Sunil[2], Neetha Thomas[3]

[1] BCA Scholar
Santhigiri College of Computer Sciences,
Vazhithala, Thodupuzha, Idukki
bcaa19_2236@santhigiricollege.com

[2] BCA Scholar
Santhigiri College of Computer Sciences,
Vazhithala, Thodupuzha, Idukki
bcaa19_2208@santhigiricollege.com

[3]Assistant Professor
Department of Computer Science
Santhigiri College of Computer Sciences,
Vazhithala, Thodupuzha, Idukki
neethathomas@santhigiricollege.com

Abstract: ***Smart Contracts have gained tremendous popularity in the past few years., to the point that billons of US Dollars are currently exchanged very day through such technology. In this paper we advocate the need for a discipline of Blockchain Software Engineering, addressing the issues posed by smart contract programming and other and other application running on blockchains. We analyze a case of study where a bug discovered in a Smart Contract Library, and perhaps "unsafe" programming, allowed an attack on Parity, a wallet application, causing the freezing of about 500K Ethers. In this study we analyze the source code of Parity and the Library, and discuss how recognized best practices could mitigate, if adopted and adapted, such detrimental software misbehavior. We also specify the Smart Contract software development, which make some of the existing approaches insufficient, and call for the definition of a specific Blockchain Software Engineering.***

Keywords: smart contracts, blockchains, software engineering.

1. Introduction

Smart contracts are becoming more and more popular nowadays. They were first conceived in 1997 and the idea was originally described by computer scientist and cryptographer Nick Szabo as a kind of digital vending machine. He described how users could input data or value and receive a finite item from a machine (in this case a real-world snack or a soft drink).

More in general, smart contracts are self-enforcing agreements, i.e. contracts, as we intend them in the real world, but expressed as a computer program whose execution enforces the terms of the contract. This is a clear shift in the paradigm: untrusted parties demand the trust on their agreement to the correct execution of a computer program. A properly designed smart contract makes possible a crow-funding platform without the need for a trusted third party in charge of administering the system. It is worth remarking that such a third party makes the system centralized, where all the trust is demanded to a single party, entity, or organisation.

Since smart contracts are stored on a blockchain, they are immutable, public and decentralised. Immutability means that when a smart contract is created, it cannot be changed again and no one will be able to tamper with the code of a contract. The decentralised model of immutable contracts implies that the execution and output of a contract is validated by each participant to the system and, therefore, no single party is in control of the money. No one could force the execution of the contract to release the funds, as this would be made invalid by the other participants to the system. Tampering with smart contracts become almost impossible.

A smart contract does not necessarily constitute a valid binding agreement at law. Some legal academics claim that smart contracts are not legal agreements, but rather means of performing obligations deriving from other agreements such as technological means for the automation of payment obligations or obligations consisting in the transfer of tokens or cryptocurrencies. Additionally, other scholars have argued that the imperative or declarative nature of programming languages can impact the legal validity of smart contracts.

With the 2015's implementation of Ethereum, based on blockchains, "smart contract" is mostly used more specifically in the sense of general-purpose computation that takes place on a blockchain or distributed ledger. The US National Institute of Standards and Technology describes a "smart contract" as a "collection of code and data (sometimes referred to as functions and state) that is deployed using cryptographically signed transactions on the blockchain network". In this interpretation, used for example by the Ethereum Foundation or IBM, a smart contract is not

necessarily related to the classical concept of a contract, but can be any kind of computer program.

A smart contract also can be regarded as a secured stored procedure as its execution and codified effects like the transfer of some value between parties are strictly enforced and cannot be manipulated, after a transaction with specific contract details is stored into a blockchain or distributed ledger. That's because the actual execution of contracts is controlled and audited by the platform, not by any arbitrary server-side programs connecting to the platform.

In this paper we advocate the need for a discipline of Blockchain Software Engineering, addressing the issues posed by smart contract programming and other applications running on blockchains. Blockchain Software Engineering will specifically need to address the novel features introduced by decentralised programming on blockchains. These will be discussed in more detail in the rest of this paper. We consider a case study, the recent attack to the Parity wallet (2017). A bug discovered in a smart contract library used by the Parity application, caused the freezing of about 500K Ethers (see [3] for a summary). We analyse the source code of Parity and the library, and reflect on the specificity of smart contract software development, noting some shortfalls of standard approaches to software development. We then discuss how recognized best practices in traditional Software Engineering could have mitigated, if adopted and adapted, such detrimental software misbehaviour. This paper aims to contribute a first step towards the definition of Blockchain Software Engineering.

2. BACKGROUND

This section presents general background information about blockchain and smart contracts technologies. It also discusses some blockchain platforms that support the development of smart contracts.

2.1. Blockchain Technology

A blockchain is a distributed database that records all transactions that have ever occurred in the blockchain network. This database is replicated and shared among the network's participants. The main feature of blockchain is that it allows untrusted participants to communicate and send transactions between each other in a secure way without the need of a trusted third party. Blockchain is an ordered list of blocks, where each block is identified by its cryptographic hash. Each block references the block that came before it, resulting in a chain of blocks. Each block consists of a set of transactions. Once a block is created and appended to the blockchain, the transactions in that block cannot be changed or reverted. This is to ensure the integrity of the transactions and to prevent double-spending problem.

Cryptocurrencies have emerged as the first generation of blockchain technology. Cryptocurrencies are basically digital currencies that are based on cryptographic techniques and peer-to-peer network. The first and most popular example of cryptocurrencies is Bitcoin. Bitcoin is an electronic payment system that allows two untrusted parties to transact digital money with each other in a secure manner without going through a middleman (e.g., a bank). Transactions that occurred in the network are verified by special nodes (called miners). Verifying a transaction means checking the sender and the content of the transaction. Miners generate a new block of transactions after solving a mathematical puzzle (called Proof of Work) and then propagate that block to the network. Other nodes in the network can validate the correctness of the generated block and only build upon it if it was generated correctly.

However, Bitcoin has limited programming capabilities to support complex transactions. Bitcoin, thus, does not support the creation of complex distributed applications on top of it.

Other blockchains such as Ethereum have emerged as the second generation of blockchain to allow building g complex distributed applications beyond the cryptocurrencies. Smart contracts, which will be discussed in the following section, are considered as the main element of this generation. Ethereum blockchain is the most popular blockchain for developing smart contracts. Ethereum is a public blockchain with a built-in Turing-complete language to allow writing any smart contract and decentralized application.

There are two types of blockchain, namely, public and private blockchain. In a public blockchain, any anonymous user can join the network, read the content of the blockchain, send a new transaction or verify the correctness of the blocks. Examples of public blockchains are Bitcoin, NXT and Ethereum. In a private blockchain, only users with permissions can join the network, write or send transactions to the blockchain. A company or a group of companies are usually responsible for giving users such permissions prior to joining the network. Examples of private blockchains are Ever ledger, Ripple and Eris.

2.2. Smart Contracts

A smart contract is executable code that runs on the blockchain to facilitate, execute and enforce the terms of an agreement. The main aim of a smart contract is to automatically execute the terms of an agreement once the specified conditions are met. Thus, smart contracts promise low transaction fees compared to traditional systems that require a trusted third party to enforce and execute the terms of an agreement. The idea of smart contracts came from Szabo in 1994. However, the idea did not see the light till the emergence of blockchain technology. A smart contract can be thought of as a system that releases digital assets to all or some of the involved parties once arbitrary pre-defined rules have been met. For instance, Alice sends X currency units to Bob, if she receives Y currency units from Carl. Immutability means that when a smart contract is created, it cannot be changed again and no one will be able to tamper with the code of a contract.The decentralised model of immutable contracts implies that the execution and output of a contract is validated by each participant to the system and, therefore, no single party is in control of the money. No one could force the execution of the contract to release the funds, as this would be

made invalid by the other participants to the system. Tampering with smart contracts becomes almost impossible.

Nodes are called miners and each one maintains a consistent copy of the ledger. Transactions are grouped together into blocks, each hash-chained with the previous block.

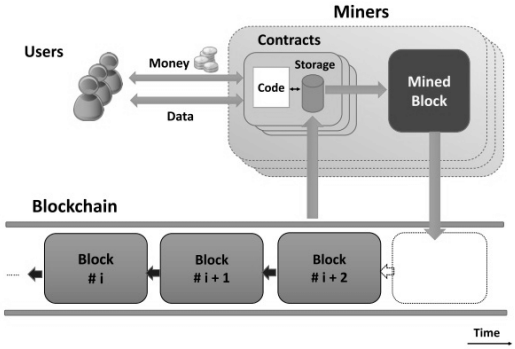

Figure 1. Smart contract system

Many different definitions of a smart contract have been discussed in the literature. In, the author classified all definitions into two categories, namely, smart contract code and smart legal contract. Smart contract code means "code that is stored, verified and executed on a blockchain". The capability of this smart contract depends entirely on the programming language used to express the contract and the features of the blockchain. Smart legal contract means code to complete or substitute legal contracts. The capability of this smart contract does not depend on the technology, but instead on legal, political and business institutions. The focus of this study will be on the first definition, which is smart contract code.

A smart contract has an account balance, a private storage and executable code. The contract's state comprises the storage and the balance of the contract. The state is stored on the blockchain and it is updated each time the contract is invoked. Figure 1 depicts the smart contract system. Each contract will be assigned to a unique address of 20 bytes. Once the contract is deployed into the blockchain, the contract code cannot be changed. To run a contract, users can simply send a transaction to the contract's address. This transaction will then be executed by every consensus node (called miners) in the network to reach a consensus on its output. The contract's state will then be updated accordingly. The contract can, based on the transaction it receives, read/write to its private storage, store money into its account balance, send/receive messages or money from users/other contracts or even create new contracts.

2.3 Ethereum Smart Contracts

A blockchain by a contract-creation transaction. A SC is identified by a contract address generated upon a successful creation transaction. A blockchain state is therefore a newly

mapping from addresses to accounts. Each SC account holds a number of virtual coins (Ether in our case), and has its own private state and storage. An Ethereum SC account hence typically holds its executable code and a state consisting of:

- private storage
- the number of virtual coins (Ether) it holds, i.e., the contract balance.

Users can transfer Ether coins using transactions, like in Bitcoin, and additionally can invoke contracts using contract invoking transactions. Conceptually, Ethereum can be viewed as a huge transaction-based state machine, where its state is updated after every transaction and stored in the blockchain. A Smart Contract's source code manipulates variables in the same way as traditional imperative programs. At the lowest level the code of an Ethereum SC is a stack-based bytecode language run by an Ethereum virtual machine (EVM) in each node. SC developers define contracts using high-level programming languages. One such language for Ethereum is Solidity [4] (a JavaScript-like language), which is compiled into EVM bytecode. Once a SC is created at an address X, it is possible to invoke it by sending a contract-invoking transaction to the address X. A contract-invoking transaction typically includes:

- payment (to the contract) for the execution (in Ether).
- input data for the invocation.

A contract-creation transaction containing the EVM bytecode for the contract in Figure 2 is sent to miners. Eventually, the transaction will be accepted in a block, and all miner will update their local copy of the blockchain: first a unique address for the contract is generated in the block, then each miner executes locally the constructor of the Puzzle contract, and a local storage is allocated in the blockchain. Finally, the EVM bytecode of the anonymous function of Puzzle (Lines 16+) is added to the storage.

To ensure fair compensation for expended computation efforts and limit the use of resources, Ethereum pays miners some fees, proportionally to the required computation. Specifically, each instruction in the Ethereum bytecode requires a pre-specified amount of gas (paid in Ether coins). When users send a contract-invoking transaction, they must specify the amount of gas provided for the execution, called gas Limit, as well as the price for each gas unit called gas Price. A miner who includes the transaction in his proposed block receives the transaction fee corresponding to the amount of gas that the execution has actually burned, multiplied by gas Price. If some execution requires more gas than gas Limit, the execution terminates with an exception, and the state is rolled back to the initial state of the execution. In this case the user pays all the gas Limit to the miner as a counter-measure against resource-exhausting attacks. Solidity, and in general high-level SC languages, are Turing complete in Ethereum. In a decentralised blockchain architecture Turing completeness may be problematic, e.g., the replicated execution of infinite loops may potentially freeze the whole network.

```
contract Puzzle {

  address public owner ;
  bool public locked ;
  uint public reward ;
  bytes32 public diff ;
  bytes public solution ;

  function Puzzle () {// constructor
    owner = msg.sender ;
    reward = msg.value ;
    locked = false ;
    diff = bytes32 (11111); // pre-defined
        difficulty
  }

  function (){ // main code , runs at every
      invocation
    if ( msg.sender == owner ) { // update reward
      if ( locked )
        throw ;
      owner.send(reward);
      reward = msg.value ;
    } else if ( msg.data.length > 0) {
      // submit a solution
      if ( locked ) throw ;
      if ( sha256 ( msg.data ) < diff ) {
        msg.sender.send(reward); // send reward
        solution = msg.data ;
        locked = true ;
      }
    }
  }
}
```

Fig. 2. Smart Contracts example.

3. STRUCTURE AND FUNCTIONALITY OF PARITY

Parity is an Ethereum client that is integrated directly into web browsers. It allows the user to access the basic Ether and token wallet functions. It is also an Ethereum GU browser that provides access to all the features of the Ethereum network, including DApps (decentralised applications). Parity also operates as an Ethereum full node, which means that the user can store and manage the blockchain on his own computer. It is a complex and critical decentralised application.

Solidity and the EVM provide three ways to call a function on a smart contract: CALL, CALL-CODE, and DELEGATECALL. The former is a call to a function that will be executed in the environment of the called contract. The other two calls execute the called code in the caller environment. Many libraries call on Ethereum are implemented with DELEGATE-CALL, typically by deploying a contract that serves as a library: the contract has functions that anyone can call, and these may be used, for instance, to make changes in the storage of the calling contracts. Solidity has some syntactic constructs which allow libraries offering DELEGATE-CALLs to be defined and" imported" by other contracts. However, at the EVM level the library construction disappears, and DELEGATECALLs and other calls are actually deployed as smart contract functionalities.

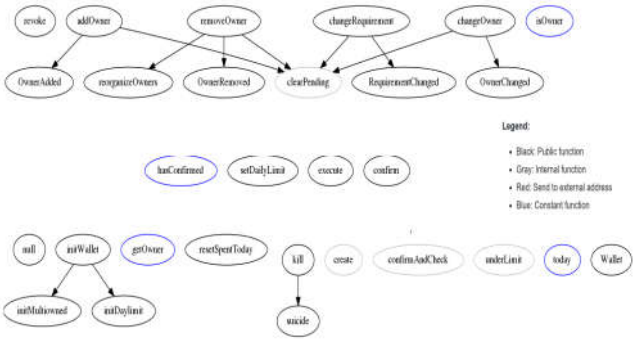

Figure 2: Parity Wallet Dependency Graph

Figure 3 shows the diagram of the functions and their dependencies for the Parity smart contract. Every call to the library will now return false and the multi-signature wallet contracts relying on the library contract code would get zero (with DELEGATE-CALL). The contracts still hold funds, but all the library code is set to zero. The multi-signature wallets are locked and the majority of the functionalities depend on the library which returns zero for every function call. The choice of defining the Wallet library as a contract instead of as a library, with the actual wallets making simple DELEGATE-CALLs to this linked smart contract, also needs to be confronted with the recommended practice of clearly defining libraries as such. Such a choice, makes the library contract behave more like a Singleton than a proper Library.

4. ROAD-MAP TO BOSE

The Parity wallet case study clearly showed that a Blockchain-Oriented Software Engineering (BOSE) is needed to define new directions to allow effective software development. New professional roles, enhanced security and reliability, modelling and verification frameworks, and specialized metrics are needed in order to drive blockchain applications to the next reliable level. At least three main areas to start addressing have been highlighted by our analysis of a specific case of study:

- Best practices and development methodology
- Design patterns
- Testing

The aim of BOSE is to create a bridge between traditional software engineering and blockchain software development, defining new ad-hoc methodologies, fault analysis patterns quality metrics, security strategies and testing approaches capable of supporting a novel and disciplined area of software engineering.

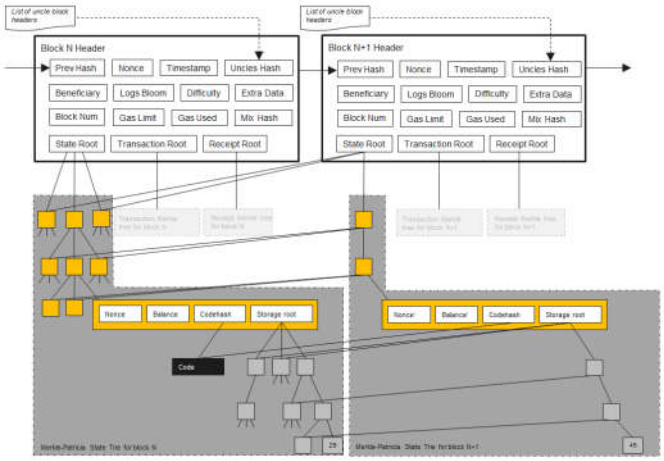

Fig. 1. Blockchain and Ethereum architecture. Each block of the chain consists of a large number of single transactions.

5. DECENTRALIZED LEDGERS

A blockchain is essentially a shared ledger that stores transactions, holding pieces of information, in a decentralized peer-to-peer network. Nodes are called miners and each one maintains a consistent copy of the ledger. Transactions are grouped together into blocks, each hash-chained with the previous block. Such a data structure is the so called blockchain, Miners use a consensus protocol in order to agree on the validity of each block, called Nakamoto Consensus Protocol. At any time, miners group their choice of incoming new transactions in a new block, which they intend to add to the public blockchain. Nakamoto consensus uses a probabilistic algorithm for electing the miner who will publish the next valid block in the blockchain. Such a miner is the one who solves a computationally demanding the graph of cryptographic puzzle. Such a procedure is called proof-of-work. All other miners verify that the new block is correctly constructed (e.g., no virtual coin is spent twice) and update their local copy of the blockchain with the new block. Bitcoin transactions essentially record the transfer of coins from one address, a wallet say, to another one. Differently, Ethereum transactions also include contract creation transactions and contract-invoking transactions. The former ones record a smart contract on the blockchain, and the latter ones cause the execution of a contract functionality (which enforces some terms of the contract). We refer the reader to the original white papers of Bitcoin and Ethereum for further details.

6. SECURITY AND SMART CONTRACTS

The smart contracts on Ethereum are generally written in high level language and then are compiled in EVM bytecodes. The most prominent and most widely adopted is Solidity, it is used even in other blockchain platforms. Solidity is a contract oriented high level programming language whose syntax is similar to Javascript.

A smart contract analysis carried out by Bartoletti and Pompianu shows that Bitcoin and Ethereum primarily focus on financial contracts. The direct handling of the assets means that the flaws are more likely to be relevant to the security and have greater financial consequences that the errors on typical applications, as evidenced by the DAO attack on Ethereum.

According to Alharby and van Moorsel, the current investigation on smart contracts has its focus on identifying and addressing the smart contract's issues and they classify them in the following four categories: codification, security, privacy and problems of performance. The technology behind Ethereum's smart contracts is still in the early stages, thus, codification and security are the most discussed topics.

6.1. Security Challenges in Ethereum

Security is the main concern when talking about Ethereum's programming owing to the following factors:

- Unknown runtime environment: Ethereum is different to the centrally administered runtime environments, either mobile, desktop or in the cloud. Developers are not used to their code being executed in a global network of anonymous nodes, without a secure relationship and with a profit reason.
- New software stack: The Ethereum stack (the Solidity compiler, the EVM, the consensus layer) is still in the developing stages, and security vulnerabilities are still being discovered.
- Highly limited ability to correct contracts: A deployed contract cannot be corrected; hence, it has to be correct before the deployment. This opposes the traditional software development process that promotes iterative techniques.
- Financially motivated anonymous attackers: In comparison with several cibernetic crimes, exploiting smart contracts offers greater incomings (cryptocurrencies' price has rapidly risen), facility for the charging (the ether and the tokens can be instantly commercialized) and a minor risk of punishment due to the anonymity and the lack of legislation on the subject matter.
- Rapid pace of development: Blockchain companies make an effort to rapidly launch their products, usually at the expense of the security. Sub-optimal high-level language: Some investigations claim that Solidity as itself leads the developers to unsecure development techniques.

6.2. Design Challenges and Patterns Usage

Understanding how smart contracts are used and how they are implemented could help smart contracts platforms' designers to create new domain-specific languages, which, with their designs, avoid vulnerabilities such as the ones that are being

outlined posteriorly. In addition, this knowledge could help improve the analysis techniques for smart contracts, by using

promoting the usage of contracts with specific programming patterns. To this day, little efforts have been made in the collection and categorization of patterns and the toolbox they use in an organized way. In the following bullet points, a general description of the typical design patterns that are inherently frequent or practical when talking about the codification of smart contracts.

Authorization: This pattern is used for restricting the code in accordance with the invoker's direction. The vast majority of analysed contracts verify if the invoker's direction is the same as the direction of the owner of the contract, before carrying out critical operations (for instance, sending ether, calling the method suicide or self-destruct).

Oracle: Is possible that some contracts have to acquire data outside the blockchain. The Ethereum platform does not allow the contracts to consult external sites: otherwise, the determinism of the calculations would break, due to the fact that different nodes could receive different results for the same consultation. The oracles are the interface between the contracts and the outside.

Randomization: Since the execution of the contract needs to be deterministic, all the nodes have to obtain the same numerical value when requesting a random number: this conflicts with the desired randomization requirements.

Time limitations: Many contracts require the implementation of time restrictions, for instance, for specifying when an action is allowed. All the contracts beings.

7. CONCLUSION

In this paper, we presented a study case regarding the Parity smart contract library. The problem resulted from poor programming practices that led to the situation in which an anonymous user was able to accidentally (it is not clear if he did it on purpose) freeze about 500K Ether (150M USD on November 2017). We investigated the case, analysing the chronology of the events and the source code of the smart contract library. We found that the vulnerability of the library was mainly due to a negligent programming activity rather than a problem in the Solidity language.

The vulnerability was exploited by the anonymous user in two steps. First the attacker was able to become the owner of the smart contract library (because it was created and left uninitialized), then the attacker did nothing more than calling the initialization function. After that the suicide function was called, which killed the library, leading to the situation in which it was not possible to execute functionality on the smart contracts created with the library, because all the delegate calls ended up in the dead smart contract library. This case clearly demonstrated a need for Blockchain Oriented Software Engineering in order to prevent, or mitigate such scenarios. The aim for BOSE is to pave the way for a disciplined, testable and verifiable smart contract software development.

8. REFERENCE

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[2] "Ethereum foundation. the solidity contract-oriented language." https: //github.com/ethereum/solidity., 2014.
[3] "A postmortem on the parity multi-sig library self-destruct," https://paritytech.io/ a-postmortem-on-the-parity-multi-sig-library-self-destruct/, 2017.
[4] "Ethereum foundation. ethereum original white paper." https://github. com/ethereum/wiki/wiki/White-Paper, 2014.
[5] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016, pp. 254–269.
[6] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 706–719.
[7] D. Siegel, "Understanding the dao attack," http://www.coindesk.com/ -dao-hack-journalists, 2016.
[8] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186. [Online]. Available: https://doi.org/10.1007/978-3-662-54455-6 8
[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," Commun. ACM, vol. 56, no. 2, pp. 82–90, 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795
[10] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," IEEE Trans. Software Eng., vol. 36, no. 2, pp. 226–247, 2010. [Online]. Available: https://doi.org/10.1109/TSE.2009.71

Author Profile

Bibin Baby UG scholar in BCA in Santhigiri College of Computer Sciences, Vazhithala.

Alan Sunil UG scholar in BCA in Santhigiri College of Computer Sciences, Vazhithala.

Neetha Thomas Asst. Professor in Santhigiri College of Computer Sciences, Vazhithala.