CURBING SOFTWARE PIRACY IN DEVELOPMENT ENVIRONMENT USING CODE SPLITTING, OBFUSCATION AND FINGERPRINTING TECHNIQUES

By

Aliyu Kufena MUHAMMAD

DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF PHYSICAL SCIENCES,

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA.

NOVEMBER, 2016.

CURBING SOFTWARE PIRACY IN DEVELOPMENT ENVIRONMENT USING CODE SPLITTING, OBFUSCATION AND FINGERPRINTING TECHNIQUES

By

Aliyu MUHAMMAD Kufena , B. Sc. Computer Science (ABU) 2010

M.Sc./Scien/24813/12-13

A DISSERTATION SUBMITTED TO THE SCHOOL OF POSTGRADUATE STUDIES,

AHMADU BELLO UNIVERSITY, ZARIA.

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF A DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE.

DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF PHYSICAL SCIENCES,

AHMADU BELLO UNIVERSITY, ZARIA

NIGERIA.

NOVEMBER, 2016.

Declaration

I, **Aliyu Muhammad Kufena**, declare that the work in this Dissertation entitled **Curbing Software Piracy in Development Environment Using Code Splitting, Obfuscation and Fingerprinting Techniques**, has been carried out by me in the Department of Computer Science. The information derived from the literature has been duly acknowledged in the text and a list of references provided. No part of this project Dissertation was previously presented for another degree at this or any institution.

Aliyu Muhammad Kufena	----------	----------
Name	Signature	Date

Certification

This dissertation entitled CURBING SOFTWARE PIRACY IN DEVELOPMENT ENVIRONMENT USING CODE SPLITTING, OBFUSCATION AND FINGERPRINTING TECHNIQUES by Aliyu Kufena MUHAMMAD meets the regulations governing the award of the degree of Masters in Computer Science of the Ahmadu Bello University, and is approved for its contribution to knowledge and literacy presentation.

Prof. S. B. Junaidu	----------	----------
Chairman Supervisory Committee	Signature	Date
Dr. B. I. Ahmad	----------	----------
Member, Supervisory Committee	Signature	Date
Prof. S. B. Junaidu	----------	----------
Head of Department	Signature	Date
Prof. Kabir Bala	----------	----------
Dean, School of Postgraduate	Signature	Date
_____	----------	----------
External Examiner	Signature	Date

Dedication

To my father, Late Arch. Muhammad Tanko Kufena.

Acknowledgement

Foremost, I praise the Creator, the Almighty for providing this opportunity and granting me the capability to complete this research successfully.

I would like to extend my sincere gratitude to my supervisor in person of Prof. S.B. Junaidu for his concern and the devotion of his precious time in advising, reviewing and guiding me throughout this research work. Also for the hard questions that triggered me to widen my research from various perspectives. I must project my gratitude to the Member Supervisory Committee, Dr. B. I. Ahmad for his immense and untiring contribution toward the completion of this work.

My sincere thank also goes to Mal. Muhammad Aminu Umar of NAERLS for his unforgettable piece of advice and support toward the realization of this research. I thank my fellow colleagues and friends: Mal. A.O. Abdussalmi, Umar Ibrahim Enesi, Salahuddeen Ridwan, Dr. Aliyu Salisu, Mal. Abdullahi Lwafu, Mal. Abdurrazaq, Mal. Sambo Aminu and Sani, Dr. Abubakar Yahaya, Mal. Aliyu Yakubu for their kind advices and assistances. Besides are my Friends: Mal. Alqasim Yusuf, Suleiman Zakir, Abdulqadir Abubakar and Ridwan Aminu, I am grateful.

Last but not the least, I would like to thank my entire family especially my mother for supporting me morally and spiritually throughout my life. My .Aunty Adama, Sister Khadija, Yusuf, Amina, Maryam and my Wife Safanatu for their supports. Thank you all and God bless you all.

List of Abbreviations

DBMS: Database Management System

AFL: American Fuzzy Lop

MAC: Media Access Control

IDC: International Data Corporation

R&D: Research and Development

IT: Intellectual Theft

IP: Intellectual Property

P2P: Peer-to-Peer

Abstract

Software Piracy has become a major problem for businesses and its widespread in many parts of the world leads to financial losses and harm to consumers. These problems caused the deployment of many security techniques to control piracy. A lot of researches have been done on software security techniques but unfortunately protecting software source code in development environment has been a challenge for software companies as software source codes are not yet transformed to unreadable codes. Therefore, software piracy emanating from development environments, mostly by insiders, has been a big problem to tackle because the prevalent software protection techniques were primarily developed for use at production stage. Based on the literature review, no research was found on piracy reduction in development environment. In this research work an architecture has been proposed and implemented based on design obfuscation that enables tracking pirated standalone software at development stage using the hardware and software aspects, online or offline, and tracing vulnerable PCs within a company. Code Splitting, obfuscation and fingerprinting techniques were used to design the proposed architecture. Two examination results processing software: ExamsLOGIC 2.0 and ELogicPLUS were used to test the proposed architecture for existing and new applications respectively. The result was compared with the result of the reviewed work and found to be improved. For both the two applications, it was found that the proposed architecture is feasible and effective.

Table of Contents

List of Figures

CHAPTER ONE

Introduction

1.1 Background of the Study

Software Piracy is the unauthorized use, distribution or copying of software illegally. It has become a major problem for businesses and it's widespread in many parts of the world leads to financial losses and harm to consumers (Rouse, 2005). The Business Software Alliance (BSA) and the Software Publishers Association (SPA) are organizations meant for reducing piracy worldwide. The Business Software Alliance (BSA) is the leading advocate for the global software industry before governments and in the international marketplace. It is an association of nearly 100 world-class companies that invest billions of dollars annually to create software solutions that spark the economy and improve modern life (BSA, 2011) . These organizations estimated that there are two-third illegal copies of software available for every legal copy of software sold and had piracy rates of 62 percent or higher (BSA, 2011). The 2010 BSA/IDC study demonstrates that even a modest reduction in software theft would have significant multiplier effects on the economic contribution of the packaged software industry. This makes software piracy a major problem for the software industries and to the world economy in general.

Software industries always strongly depend on copyrights and other Intellectual Property (IP) to drive innovation and ensure a return on investment in R&D. Hence, protecting their products from being pirated is one of the major tasks they consider in order to keep the company up and running. With the improvement in technology over decades, software vulnerabilities such

as Reverse Engineering increase at almost the same pace with technology. For almost every software protection technique there is an equivalent hacking technique for it.

In the case of packaged software, it is common to find counterfeit copies of CDs incorporating the software programs, as well as related packaging, manuals, license agreements, labels, registration cards and security features. Counterfeiting is a serious problem for the software industry, as advances in technology have enabled a growing number and variety of commercial enterprises to manufacture and distribute counterfeit software on a massive scale. This is because the risks of being caught are relatively low and penalties are far less than for engaging in other illegal activities (BSA, 2011).

Besides, Software Piracy also serves as one of the major ways of distributing worms by hackers. Although some consumers may think they are getting a great deal when they obtain pirated software, it is more likely they will receive a substandard product with hidden cyber security threats. The fact is that using illegal software puts consumers' personal information, financial security, and even reputation at risk. At the very least, it can lead to software incompatibility and viruses, drive up maintenance costs, and leave consumers without technical support or security updates.

It is common for sites that offer access to pirated software and piracy-related tools to distribute malicious code that damage IT security and performance. Indeed, a significant percentage of counterfeit software or key generators downloaded from P2P sites contains malicious or unwanted code. In an IDC study, research revealed that one in four websites that offered pirated software or counterfeit activation keys attempt to install infectious computer code, like Trojan horses and key loggers, on test computers. The study found that 59% of counterfeit software or key generators downloaded from P2P sites contained malicious or

unwanted code. A subsequent study by Harrison Group Inc. found that companies using unlicensed or counterfeit software were over 70% more likely to have critical computer failures lasting 24 hours or more and/or experience the loss or damage of sensitive data (BSA, 2011).

These are some of the problems that urge companies to devise and improve ways of protecting their products from reverse engineering, counterfeiting, and intellectual theft. Some of the adopted standard methods for protecting software are Code Obfuscation, Software Tamperproofing, Software Birthmarking, Software fingerprinting and watermarking (Collberg and Thomborson, 2002).

In this research, controlling software piracy in the development environment is the major task to be accomplished.

1.2 Problem Statement

Source codes of a software company are available on the developers' computer whereby maintenances and updates are done. At this level, software and developers' PCs are vulnerable to attacks and piracy to claim ownership or use as worm distributors. Hence, preventing the source codes at this stage is a challenge to companies. Also, tracing the code owner's pirated copy within the firm becomes a hectic task.

1.3 Research Motivation

In a company where the source codes of their software product are available on their developers' system, what are the protective measures to be taken to protect the source codes from being pirated? How can a software company trace and validate its pirated software code that is in use without totally obfuscating the source code?

To answer the research questions, there is a need to identify the program's high level code and explore the existing security techniques.

1.4 Aim and Objectives

This research aims at designing and implementing an architecture to enable tracking and disabling pirated software instances via online or offline and also tracing its source in a development environment.

The objectives of the research to be achieved are to:

1. Design an architecture that allows tracking pirated standalone software at development stage using the hardware and software aspects via online or offline
2. Implement the proposed architecture using existing and new application scenarios
3. Evaluate the system to demonstrate the architecture piracy tracking capability in both scenarios using testing software

1.5 Research Methodology

The following procedures were adopted in the execution of this research:

1. Review previous works on software piracy, software protection techniques and related works
2. Design the architecture by applying the code-splitting technique on an arbitrary program $\mathcal{P}$
3. Choose an obfuscator with multi-level code hiding
4. Obfuscate a segment of the program, called *Chip* (Partial program obfuscation)
5. Configure the online Tracking Server

6. Implement the Tracking Server database using *mysql* as DBMS and PHP as the programming language

7. Evaluate the system's capability to detect pirated copy of a software on different systems or PCs, online or offline by choosing some applications for testing

1.6 Scope and Limitation

a. The quality of the transformer(s) $T()$ used is not put into consideration
b. If the tracking mode is set to "Online", the pirated application is tracked by the server only if the System is online
c. The obfuscators used are for countering only static reverse engineering attacks
d. Security on system MAC address was not put into consideration

1.7 Organization of the Dissertation

This dissertation consists of five chapters, of which Chapter Two consists of reviewed work on Software Piracy, Software development stages, Code Obfuscation with related works, summary of their limitations of the reviewed works and how they differ from our research. Chapter Three consists of the architectural design, equation derivations, obfuscation software used, scenarios considered and applications used for testing the feasibility of the proposed architecture. Chapter Four encompasses the implementation of the proposed approach, evaluation of the proposed system in comparison with previous researches. Summary of the work done, recommendations for future research and conclusions were reflected in Chapter Five.

CHAPTER TWO

Literature Review

2.1 Introduction to Software Piracy

Piracy of intellectual property (IP) is equivalent to the theft of tangible goods. Piracy can take several forms depending on the type of intellectual or information and the technological access to it (Fischer and Andrés, 2005). Besides Software piracy is not only about Intellectual theft but also a factor that destroys economy. In particular it is a big challenge to the companies in particular and the world in general.

In some cases, even culture is involved, where societies see intellectual property as a common good and not the property of its developer. By the end of 2007, there were more than one billion PCs installed around the world; nearly half have pirated software on them. With more PCs being shipped into emerging markets, lowering that percentage will be a long-term challenge.

Dealing with software piracy in emerging markets is still a challenge. Rapid growth in first-time users from the high piracy consumer and small-business sectors affects country averages even when piracy drops in other areas. Though there was notable progress in the battle against PC software piracy in 2007. Out of the 108 individual countries studied in BSA and IDC Global Software Piracy Study report, the piracy rate dropped in sixty-seven countries from 2006 to 2007 and increased in only eight countries (BSA, 2007).

In legal terms, intellectual property covers three distinct sets of rights: copyrights, patents, and trademarks (Asongu and Andrés, 2012).

2.2 Methods of Protecting Software

There are three major types of software threats: Reverse Engineering, Tampering, and Piracy. Obfuscation, Tamperproofing and Watermarking are the respective techniques used to curb the existing threats (Collberg and Thomborson, 2002). Software reverse engineering is a technique to inspect the inner workings of software applications, tampering covers ways to modify a software while piracy concerns unauthorized use of software.

2.2.1 Architectural Protection Techniques

Based on the architectural design of a system of an application, a certain level of protection can be achieved. Below are some of the existing architectures used in protecting software by Cappaert (2012).

a. Client-Server Solutions

This is one of the earliest and commonly used techniques used in protecting software. It an approach in which the services are distributed to clients instead of the software. In most cases no software protection technique is used to protect the source code of the software as the software is hosted on the server. Protecting the server is the major challenge as it encompasses Network, Hardware and Software security like the server Operating System itself. The approach is sometimes referred to as “software as a service”. The source code or executables are always hosted on the server side as shown in Fig. 2.1. This setup makes the server to be seen by an attacker as a *Black Box* that only requires an input from client and returns back response. Although network bandwidth may be the major drawback in relation to this Architecture which may lead to rendering the server temporarily unavailable. This problem may be solved by upgrading the Network infrastructures and Mirroring.

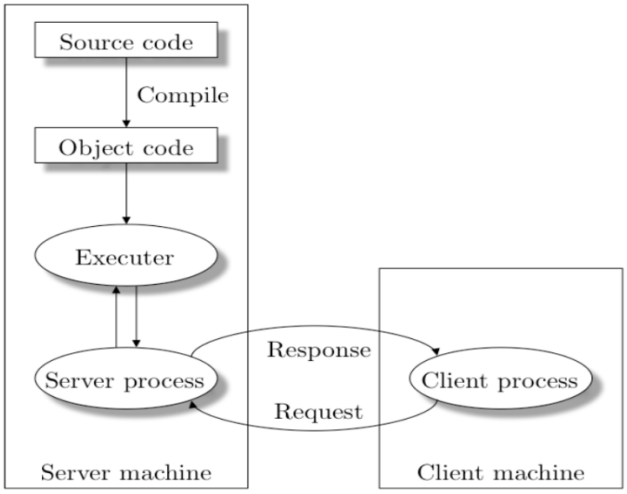

Fig. 2.1: The Client-server Model *(Cappaert, 2012)*

b. Partial Client-Server Solutions

This model splits source code into a *critical* and *non-critical* parts as shown on Fig. 2.2. The critical part is residing on the server side while the non-critical part runs at the individual clients' side.

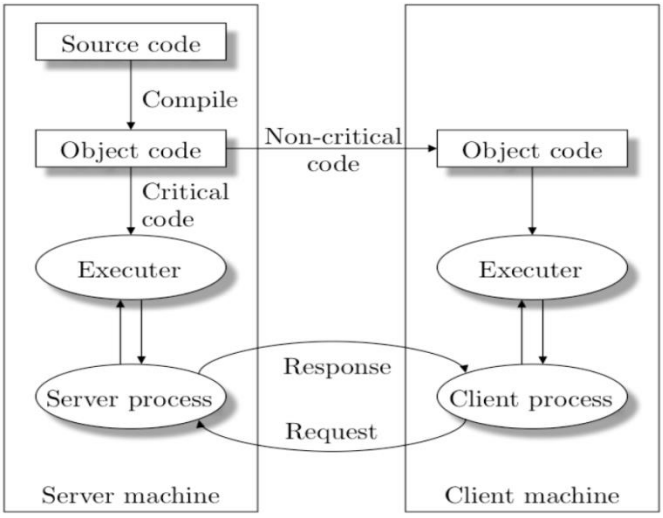

Fig. 2.2: The Partial Client-server Model *(Cappaert, 2012)*

The security of the *critical part* in this model also depends on the server security. Service load and the software source code are shared between the server and the client machines which with

this communication between the *critical* (server) and *non-critical* (client) may result to a bandwidth problem.

2.3 Concept of Obfuscation

To obfuscate simply means: to render obscure, darken or confuse. Code obfuscation technique obscures the control, data, layout, or design of the software's original implementation and give a semantically same but new implementation of the target software that make reverse engineering much harder (Das, 2014). An obfuscator is a tool which through the application of code transformations converts a program into an equivalent one that is more difficult to reverse engineer. From another view obfuscation is seen as a process of breaking abstractions and unstructuring data structures (Collberg *et al.*, 2000).

More formally, an obfuscator $\mathcal{O}$ takes a program $\mathcal{P}$ and produces a program ($\mathcal{P}'$) that has the same functionality as $\mathcal{P}$, yet it is *unintelligible* in some sense. It is assumed an obfuscator should provide a virtual black-box view to the user such that a $\mathcal{O}$ ($\mathcal{P}$) output is only seen by the user as equivalent to that of the original program $\mathcal{P}$ though a perfect obfuscator has been proven impossible to create (Barak *et al.*, 2001). It is still possible to obfuscate a program in such a way that it is very expensive to reverse engineer (Janssen, 2016).

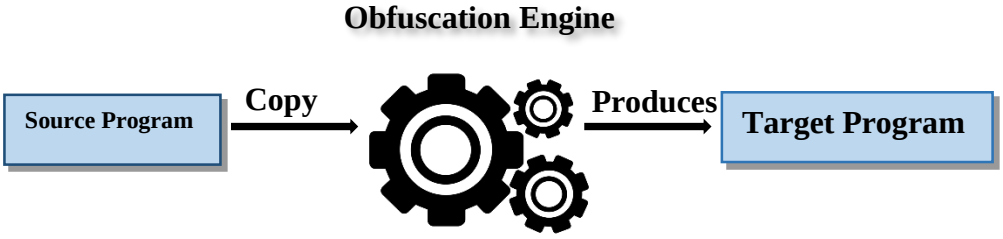

Fig. 2.3: Working Principle of Obfuscators

Code obfuscators receive the source code address or directory and the destination as parameters to save the resulting obfuscated code. Some provide obfuscation options or parameters like variable and constants' name replacement, encoding, encoding Key, optimization options, locking to an IP Address (es) or domains, Expiring Date assignments and so on. After receiving the parameters, the obfuscator reads the source code and then apply the actual underlying obfuscation algorithm on it and generate the obfuscated code which is saved in the specified destination directory as shown in Fig. 2.3. Besides the original source code remains unchanged and kept for future updates.

According to Collberg and Thomborson (2002), if a set of obfuscating transformations $\mathcal{T} = \{\mathcal{T}_1,\ldots, \mathcal{T}_n\}$, and a program $\mathcal{P}$ consisting of source code objects (classes, methods, statements, etc.) $\{S_1,\ldots,S_k\}$, that produce a new program $\mathcal{P}' = \{\ldots, S'_j = \mathcal{T}_i(S_j),\ldots,\}$, such that:

a. $\mathcal{P}'$ has the same observable behavior as $\mathcal{P}$, meaning the transformations *preserve the semantics* of the initial program $\mathcal{P}$
b. The *obscurity* of $\mathcal{P}'$ is maximized such that clearness and reverse engineering $\mathcal{P}'$ consumes more time than $\mathcal{P}$
c. The *resilience* of each transformation $\mathcal{T}_i(S_j)$ is maximized, by making it difficult or time consuming to automate reverse engineering of the transformation
d. The *stealth* of each transformation $\mathcal{T}_i(S_j)$ is maximized, i.e. the statistical properties of S'_j are similar to those of S_j
e. The *cost* of execution penalty that the transformation incurs in $\mathcal{P}'$ based on time or space which should be minimized

From the above listed characteristics as shown in Fig. 2.4, the three parameters for measuring quality of an obfuscation technique are:

a. Potency: evaluates the strength of the obfuscating transformation added to P against humans' ability to understand. It is the measure of "how well are the functions hidden from manual analysis?"
b. Resilience: Deals with how well the obfuscation avoids automated analysis
c. Cost: This is the measure of the resource overhead that was added after transforming the initial program

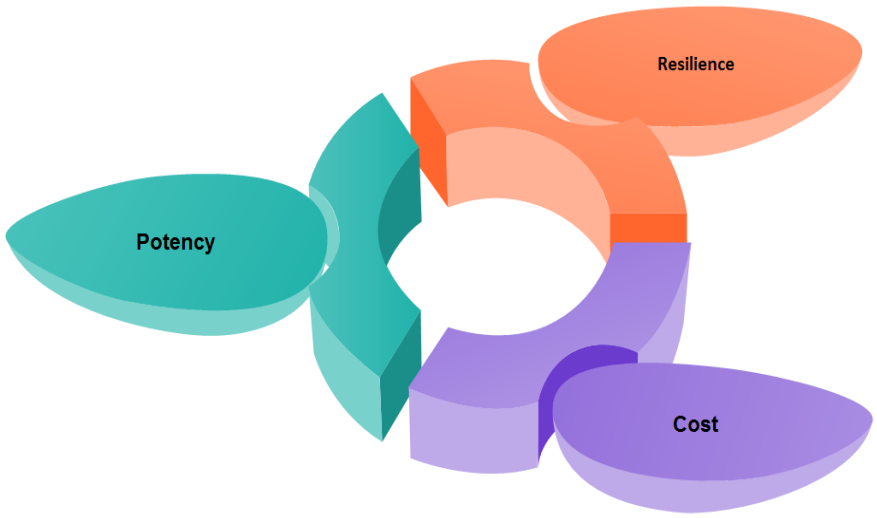

Fig. 2.4: Metrics for measuring quality of an obfuscation technique

There are many proof-of-concept obfuscators developed in academic research that are not released as a full-featured project, and are thus not yet for any production use. Besides in many available commercial obfuscators implementation, the exact working principle is not published because of general 'security through obscurity'.

Obfuscator-LLVM is an open source project that was developed by Junod. Tigress was also originally developed by Collberg at the University of Arizona. While the project is not an

open source, pre-compiled binaries are available free of charge. It works directly on the C source code instead of an intermediate representation because it performs a source-to-source transformations (Janssen, 2016). PHPLockIt!, PHP Obfuscator, SourceCop, srcProtector for PHP, Zend Guard: Encoding and obfuscation and many more obfuscators are available.

2.3.1 Classification of Obfuscation

Over decades, many techniques and tools have been developed in trying to improve software obfuscation in a way that makes the process of reverse engineering as difficult as possible.

Basically there are four main classes of code obfuscation transformations: Layout obfuscation, Data obfuscation, Control obfuscation and Preventive transformation.

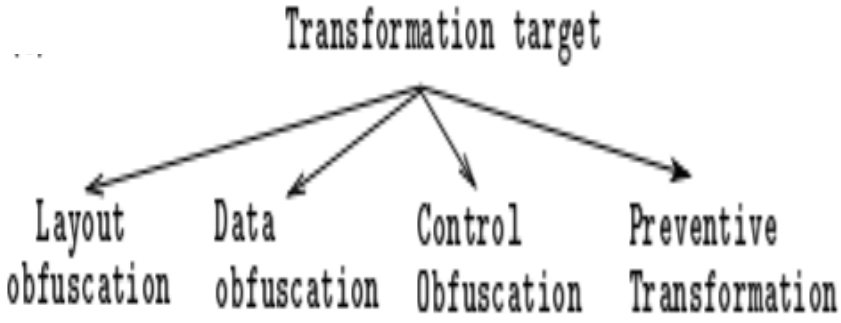

Fig. 2.5: Basic classifications of Obfuscation (Collberg *et al*., 1997)

The basic obfuscation classifications are:

a. Layout or Lexical Obfuscation: It refers to obscuring of the software layout by changing format of the source code, increasing and decreasing nesting or arrays' dimension, variable and functions' renaming, insertion of death code, deleting comments or removal of debugging information through complicating the lexical structure of the program.

b. Data Obfuscation: It prevents the extraction of information from data. The techniques under this are: array splitting, variable splitting, changing the scope and lifetime of data. This method is applied for data not code. It includes obfuscating any hardcoded value in a code. Most obfuscation transformations are not one-way. However, a few transformations are considered to be one-way.

c. Control Obfuscation: This refers to the obscuring of the control flow of the program. This kind of obfuscation technique is mainly based on self-modifying code.

d. Preventive Transformation: This approach mitigates the program such that the code itself will force the debugger or disassembler to fail, this depends on debuggers' or disassemblers' weaknesses. This is done by inserting "junk bytes" in between other instructions and the disassemblers interpret these bytes as the beginning of an instruction.

Besides the above classifications, merging, splitting of code sections or classes and type hiding are possibly grouped under *design Obfuscation* which helps in obscuring the design of the target program (Das, 2014). Fig. 2.6 shows further classifications of obfuscation with examples by Collberg *et al.* (1997) taxonomy.

This research is based on design and layout obfuscation. As code splitting falls under design obfuscation and the obfuscators used are layout based.

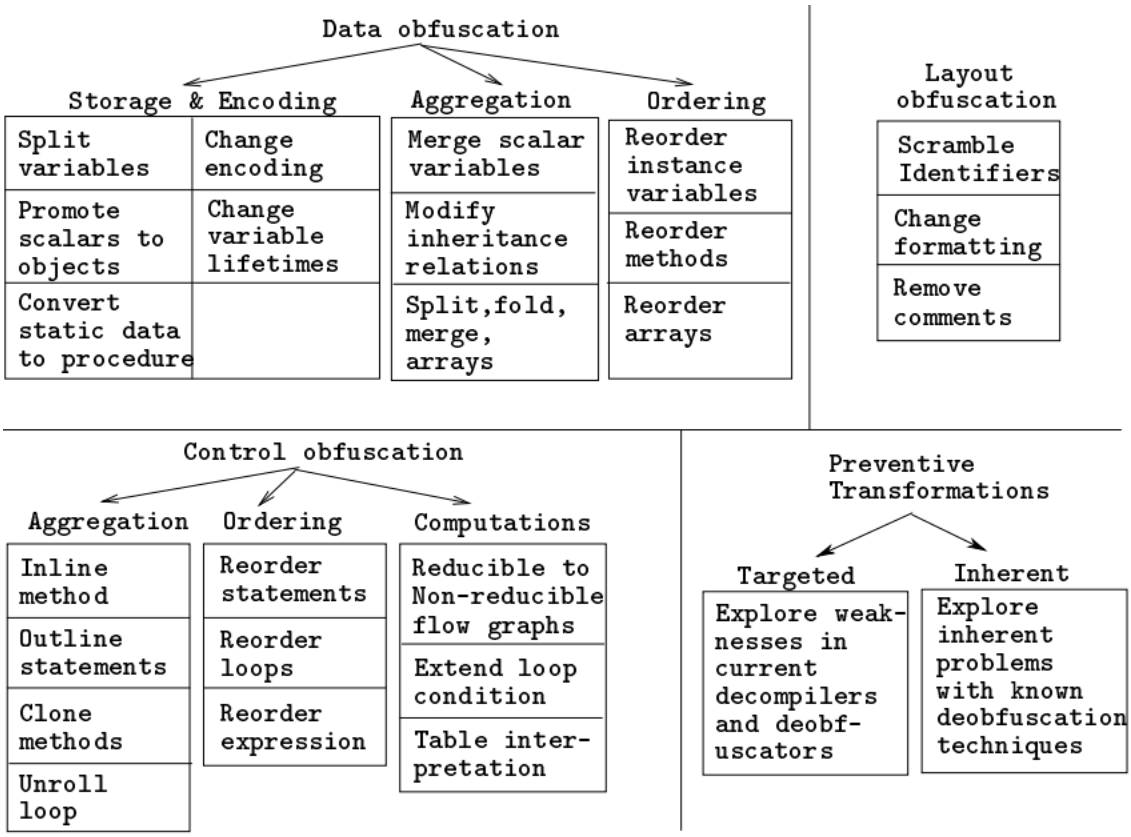

Fig. 2.6: Classification of Obfuscating Transformation (Collberg *et al*., 1997)

2.4 Reverse Engineering

This is the process of investigating and understanding a software program's binary code and recreating it so as to trace the original source code of the software. It is also used to uniquely identify the system's modules and relationships between them and create a representation of the target system in another form. Depending on the adversary's intention, hidden algorithms, secret keys, and other information about the software may be the primary target of the adversary (Richa, 2014).

2.4.1 Static Analysis

This is a technique use on non-executing source code of a software which consists of static disassembling and a subsequent static examination steps. It deals with understanding of the

binary code of a software (Janssen, 2016). *Linear sweep* or *recursive traversal* are used in disassembling a code. Scanning over the code is the task accomplished in a linear sweep.

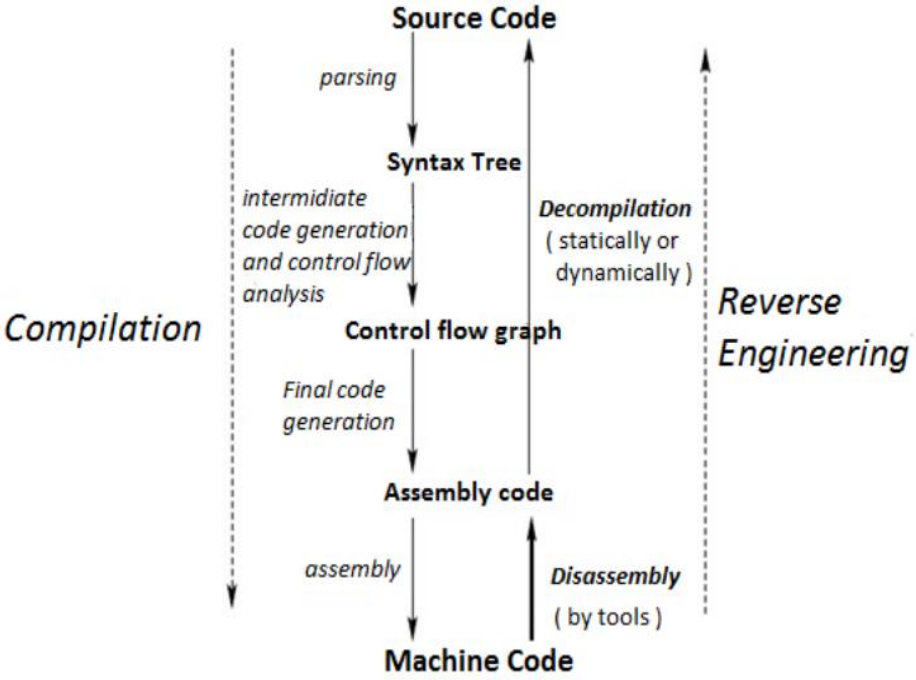

Fig. 2.7 : Stages in Reverse Engineering *(Cappaert, 2012)*

Instructions are disassembled and assume each instruction is followed by another instruction. (Cappaert, 2012).

2.4.2 Dynamic Analysis:

This is a technique used on executing code. It consists of tracing the contents of registers, executed instructions, data values, and so on. Static attack is less powerful, less time consuming and less complicated than a dynamic attack. Emulators, debuggers like SoftICE and IDA Pro are prominently used in this type of reverse engineering technique. Fig. 2.7 gives the stages of reverse engineering a software.

2.5 Code Splitting Technique

Code Splitting is an obfuscation technique in which a software code is split into two or more parts to attain some level of obscurity based on a particular splitting rule as shown in Fig. 2.8. This splitting can be based on classes as mentioned earlier in section 2.4. Code Splitting falls under Design Obfuscation which deals with obscuring the design related information of a software. The means of communication or linkage among the split resulting programs: Program 1, Program 2, Program 3,…, Program n is also defined by the splitting rule. Though Das (2014) did not clearly specify the rule used to split the source program but he clearly specified the means of communication between the split programs, which is via a shared memory.

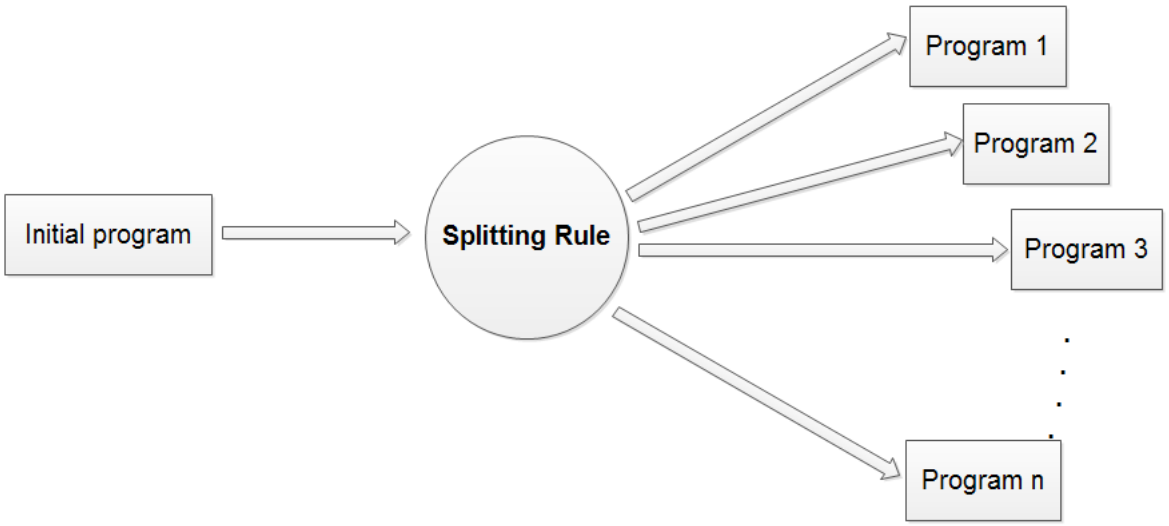

Fig. 2.8: Code Splitting Process

Based on the classification of Collberg *et al*. (1997), the Branch Insertion transformation under dead or irrelevant code insertion is a closer approach to code splitting than the remaining transformations. Considering Fig. 2.9, where S= S_1 ;…; S_n is a basic block in Fig. 2.9 (a) a predicate(opaque), P^T which always returns true is inserted into S which was split into half,

S_1 ;…; S_j and S_{j+1} ;…; S_n. Such that P^T in the program serves as an irrelevant code. In Fig. 2.9 (b), S is also split as in Fig. 2.9 (a) into S_1 ;…; S_j and S_{j+1} ;…; S_n but the second half, S_{j+1} ;…; S_n is duplicated and obfuscated using two different transformations to produce $\boldsymbol{S^a}$ and $\boldsymbol{S^b}$ as the respective target programs from two different transformations. A predicate ($\boldsymbol{P^?}$) that evaluates to either TRUE or FALSE at runtime is inserted into the code. Fig. 2.9 (c) shows the use of two different sets of obfuscating transformations albeit introducing bug in the second half of the transformed code, $\boldsymbol{S^b}$, which makes $\boldsymbol{f(S^a) \neq f(S^b)}$. So the inserted predicate is set to always choose the right code to run. These three approaches make the code hard to be understood by an adversary and use the idea of splitting codes.

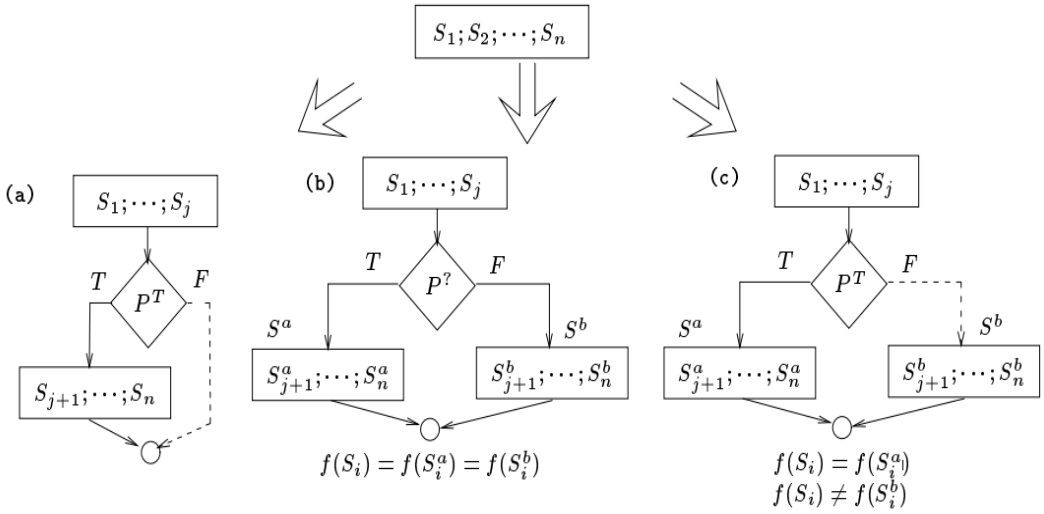

Fig. 2.9: The Branch Insertion transformation (Collberg *et al*., 1997)

2.6 Software Watermarking and Fingerprinting

Technically, software watermarking and fingerprinting are not used to prevent analysis, tampering, or piracy. The major aim is to create fear in the attacker for being caught. Software watermarking is an overall term that covers all techniques that prove ownership of software.

Software fingerprinting deals with mapping an instance of a program to a unique user. In this case, this type of watermarking gives room for tracing back a pirated copy to the user who violated the license agreement by illegally distributing his copy (Cappaert, 2012). Therefore, fingerprinting technique will be used in this research to trace pirates of software.

2.7 Software Development Stages

There are General Software Process Models like Waterfall model, Agile, Prototype model, Evolutionary development model and Iterative model that are considered in software development. Having a top level view of the models, we see two major phases: The Development environment and the Production Environment.

Considering the Waterfall model which serves as a baseline for many other lifecycle models (AliMunassar and Govardhan, 2010) with the stages: Requirement gathering, System Design, Implementation, Testing, deployment and System maintenance as shown in Fig.2.10. With the exception of the last two phases, all the remaining phases are at development stage while the deployment and maintenance phases are at production stage. This simply implies that the users at development and production stages are respectively the developers and customers or consumers of the software.

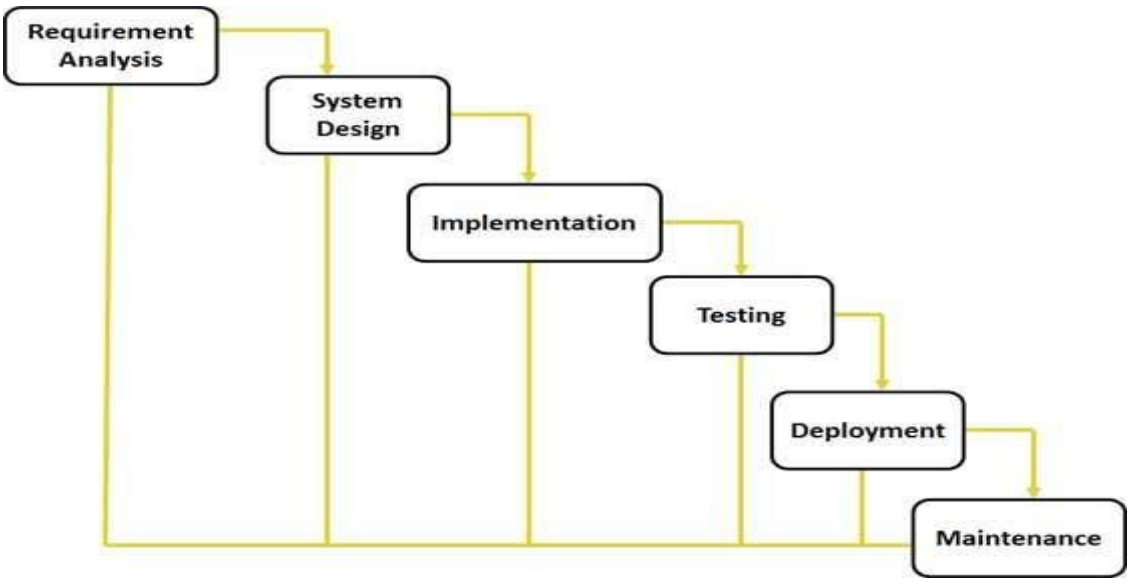

Fig. 2.10 : Waterfall Model phases *(SDLC, (n.d.))*

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are tons of models, and many companies adopt their own, but they all have similar patterns (Lewallen, 2005). The general, basic model that gives a top level representation of the models putting into consideration the working environment for each phase is depicted in Fig. 2.11.

Development Environment **Production Environment**

Users: Developers Users: Customers

Fig. 2.11: General Software life cycle model

Moreover software transition from development to production environment causes some recommended settings that vary between the two environments. Taking as an example in WAMP

server: PHP settings, the settings for displaying error differs in the "*php.in*" file with three recommended values of the *display_errors* variable with respect to the software environment as shown in Table 2.1.

Table 2.1: Values Variation in Some php variables in different Environments

Variables	Default values	Development Environment	Production Environment
display_errors	On	On	Off
display_startup_errors	Off	On	Off
html_errors	On	On	Off
log_errors	Off	On	On
short_open_tag	On	Off	Off

In development environment, the variable *display_errors* is set to "On" for the developer to see errors and warnings that are to be fixed in the software otherwise it is set to "Off" at production environment. Few of the variables are listed in Table 2.1 under *Quick Reference* which are the settings that differ in either the production or development versions with respect to PHP's default behavior.

2.8 Related Literature

There have been a considerable amount of research work on software obfuscation with diverse methods of implementation as in the literature below:

Collberg et *al*. (1997) gave a detailed classification of obfuscating transformation as mentioned earlier. They derived three basic metrics: potency, resilience and cost in measuring the effect of an obfuscating transformation. They explored many different obfuscation classifications and some properties.

Myles (2006) proposed a novel scheme in both software watermarking and birthmarking techniques that provides a more significant improvement compared to the existing watermarking and birthmarking techniques. The techniques demonstrate a stronger balance between the properties of credibility and resistance to transformation. The improved watermarking is the first known dynamic watermarking. When an embedded watermark has been destroyed the instances of the program are easily identify as the degree of transformation is high and also detect individual occurrences of software theft. Branch-Based software Watermarking (Authorship Mark, fingerprintmark and temper Detection) and Dynamic Birthmarking.

Cappaert (2012), worked on two techniques to prevent both static and dynamic attacks. In case of the first technique for preventing static attack the technique makes it difficult to statically derive the control flow of a program based on one-way functions to make backward analysis hard, and bijective functions that do not leak any control flow information that can assist an attacker. They have used three application models and several attacks to evaluate the strength of his technique. With a second technique to stop dynamic attacks: Dynamic analysis of the code and dynamic code tampering.

Das (2014) used the code splitting approach to curb dynamic reverse engineering by splitting a program into three parts: *Server*, *Client-Start-up* and *Client* programs, in which both *Server* and *Client* programs communicate through a shared memory created by the *Server*. Only a user interacts with the *Server* program. The *Server* creates shared memory and starts up the *Client-Start-up* which in turn the *Client-Start-up* starts the *Client*. The *Client-Start-up* does start the *Client* only and halts after starting the *Client*. Eventually, only the *Server* and the *Client* continue running and communicating via the created shared memory. They used self-modifying code technique taking advantage of some operating system functionalities to handle the static

attack and the obfuscation method was used to prevent a dynamic technique. The approaches he used were based on design and Preventive Transformation. The client acts as a zombie with process id 1. Hence it attaches itself to the shared memory location for data exchange. It finally terminates successfully. Rules for splitting the software code was not explicitly explained at code level, the hardware aspect was not put into consideration also.

Janssen (2016) combined active learning and fuzzing to automatically learn state machine models of obfuscated software, which gives an idea of the behavior of the program. AFL fuzzer was used to create test cases or semi-valid input data for a target program in combination with the active learning system. Active learning used automatically learn state machine models of obfuscated software. The target program is then run with the generated input data, and any test cases that cause the program to unexpectedly crash are saved for later inspection. They also used a genetic algorithm to rank test cases.

Pomilia (2016) proposed and implemented a framework with the aid of some tools to obfuscate and run a test on large number of Android malware. Using their proposed framework they obfuscated malware from 2012 to 2014. They used *Androguard* to analyze a malware belonging to the *Opfake* family detecting the obfuscation techniques used to obfuscate the malware and trying to understand what they hide. Based on their results they concluded that anti-malware tools have not improved in their capabilities within 2012 to 2014.

2.9 Gap in the Reviewed Literature

In all the previous literature we reviewed, obfuscation, code splitting, fingerprinting and other standard techniques were used to trace a vulnerable computer at software production stage, as protection techniques were primarily made for use at the stage. Securing a software and

tracing a vulnerable computer at the Development Stage were not put into consideration which are the major problems we are focusing on, in this research. Besides, the splitting rule in code splitting technique was not clearly defined during the process of applying the technique.

CHAPTER THREE

System Analysis and Design

3.1 Introduction

This chapter spells out the detailed architectural design, components of the system, the programming language used to implement the proposed system, the obfuscators used, scenarios to use the architecture, testing applications used, company approach to produce the components in bulk, approaches or methods used in implementing the proposed architecture.

3.2 Split Components

a. **Function Definitions (f_D):** It is a file that keeps the functions definition of an application

b. **Function Calls (f_C):** It is a file that serves as a *workspace* for the developer where calls of the defined functions in f_D are made. Both f_D and f_C are viewed as sets.

c. ***Chip*:** It is a program that keeps the user's credentials as registered on the *Tracker* and it does communicate to the Tracker. It also serves as a connector that links f_D and *fc*.

3.2.1 *Chip* Parameters

a. Registered PC's MAC Address

b. Registered User ID

c. Relative f_D path, it can either be Offline or Online(Intranet)

d. Decision String

e. Response Contact (email or Phone Number) [optional]

3.2.2 f_D Parameters

a. Add and call the *secure* function

b. *Add* relative *Chip* path in *secured* function

3.2.3 f_C Parameters

a. Relative database *connection_file(db.php)* path [for existing Software]

b. Relative *Chip* path

3.3 System Architectural Design

There are existing online version control code management systems like *Bitbucket* cloud which works with a *Mercurial* repository. It uses synchronization approach to update client's code but pulled source codes are not protected, as authentication is done only when synchronizing software code not after. This user authentications still leaves the codes unprotected after the code synchronization.

The concept behind the proposed architecture is based on Design and Lexical obfuscation where an application is split into two major parts: function definitions and function calls. Fig. 3.1 depicts the users and connections among the components of the system. It shows the necessary components a user or developer should have and the dependencies of some components. f_D depends on *Chip* and vise vasa. Dependency is represented by a yellow arrow. The Administrator does the users' registration on the server using their respective PCs' MAC Addresses and the software ID generated for each instance of the software which are all passed to the *Chip* as parameters during *Chip* production.

3.4 System Components

The system is made up of different components as explained below:

a. Documentation: The documentation consists of the functions' synopsis which includes: function ID, name, signature, return value, comment, date created, last date updated and

ID of the developer who created the function. It consists of all collated functions definition from the users. Users pull the completed documentation onto their PCs for future or offline references.

b. Users: These are the programmers working on an application.

c. UserLib: It consists of the functions defined by individual users and their their respective documentations of the functions saved as ***UserDoc***. It is defined by a user as shown in Fig. 3.1. For each user, both the *Chip* and f_D are obfuscated with a two way file reference between the files but f_C is not obfuscated but it references the *Chip*.

d. Administrator: The basic reports generated by the Administrator are reports on registered users and responses based on tracked valid and invalid users. The updated source code of the *Chip*, f_D and its documentation are kept on the server (which serves as a repository), for easy code management and access by users. The f_D and its documentation are fed or updated from the individual *UserLib* and the *UserDoc*. The Administrator also updates the collective documentation to be pulled by users.

3.5 System View Based on the Partial Client-server Architecture: In Section 2.2.1, security based on architectural implementations were discussed. Fig. 3.2 depicts the proposed architecture based on the *critical* and *non-critical* code splitting in which the critical part of the code are the *Chip* and $\boldsymbol{f_D}$ residing on the server and the *non-critical* part of the code is the $\boldsymbol{f_C}$.

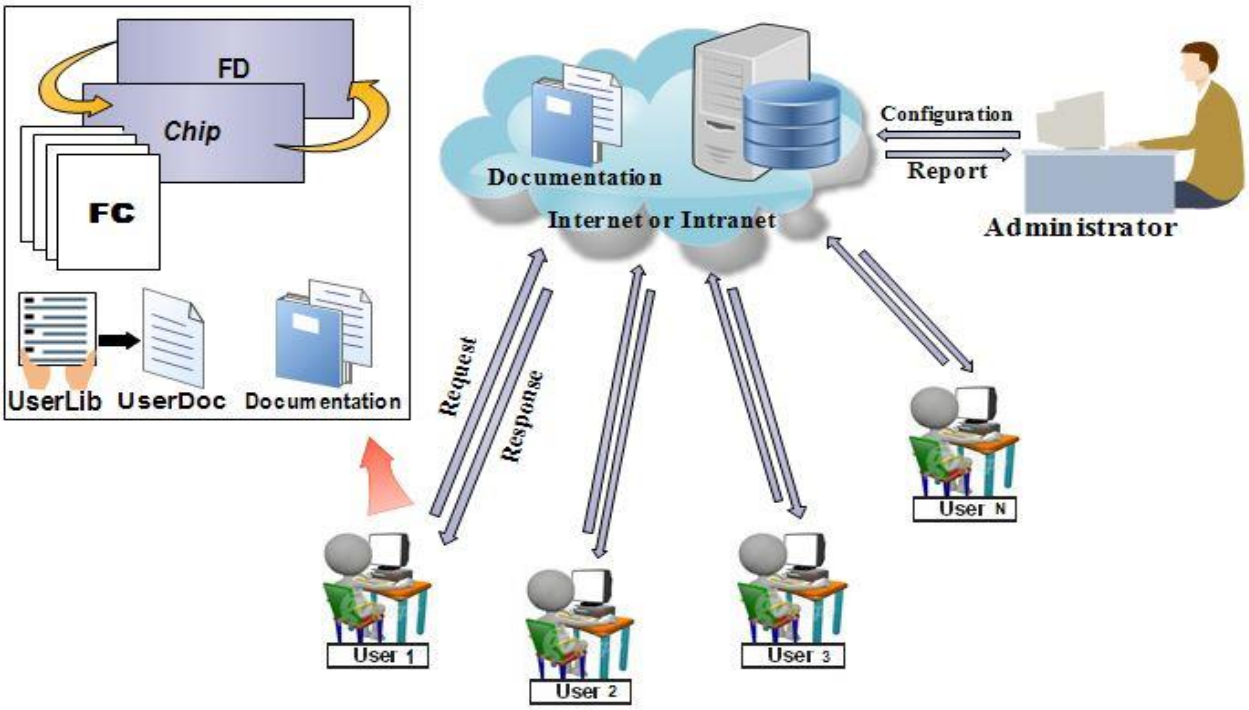

Fig. 3.1: An Overview of the Architecture

Moreover, in the proposed architecture both the two code sections are on the user or client machine. Hence, only some few services are rendered by the server to the client.

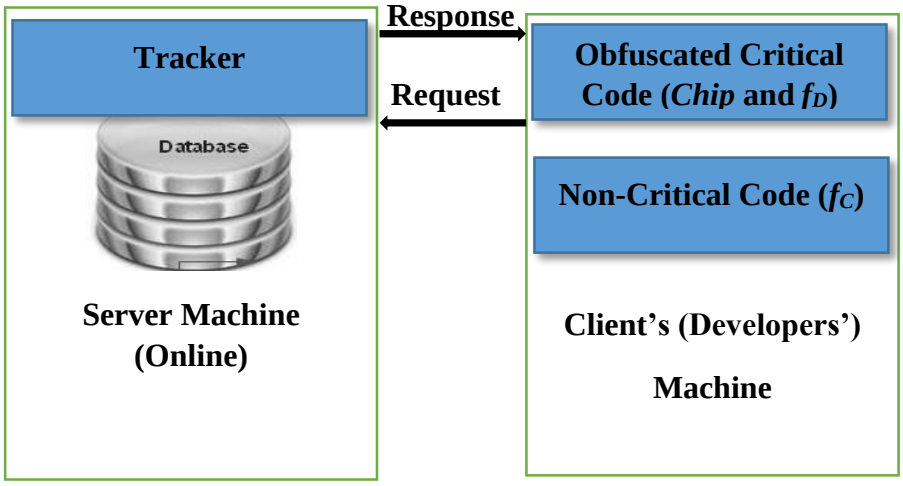

Fig. 3.2: Proposed Architectural design based on partial client-server terminologies

3.6 Fingerprinting Technique Used

Software pirates can be traced using fingerprint with the aid of obfuscation (Collberg *et al.*, 1997). Based on this we embedded a fingerprint before obfuscating the source code. By including the fingerprinting technique, we put into consideration both the software and hardware aspects. We mapped the System MAC address to the unique fingerprint (made up of software ID and User ID) on the server to combine both Hardware and Software techniques. This is because a truly software technique that is meant to control software piracy requires employing a technique that combines both hardware and software-based approaches (Myles, 2006).

In this research we generate each instance of the application fingerprint based on: Code of the application, its version and the user's ID. This is done by the tracker during *Chip* production.

3.7 System Modeling (Derivation)

Assuming $\boldsymbol{\mathcal{P}}$ is the source program designed based on the proposed architecture of function definitions ($\boldsymbol{f_D}$) and calls($\boldsymbol{f_C}$) to enable program splitting into definitions and calls or Class definition (C_D) and Objects' instantiation (O_I), for programs that were designed based on Object Oriented approach as in Equation 3.1.

$\mathcal{P}$= source program

$$\mathcal{P} = f_D + f_C \quad or \quad \mathbf{OR} \quad [\mathbf{C_D + O_I}] \text{ (Code Splitting)} \tag{3.1}$$

A new program called the *Chip* is injected into the program $\boldsymbol{\mathcal{P}}$, after passing the Chip parameters listed in Section 3.2 to the Chip and also to the components: f_D, and f_C as specified in Section 3.2. Addition of the *Chip* produces a new modified program $\boldsymbol{\mathcal{P}'}$ as shown in equation 3.2.

$$\mathcal{P}' = f_D + Chip + f_C \tag{3.2}$$

Where f_D and *Chip* check for their existences and appropriate parameters are passed to f_D , *Chip and* f_C. Checking simply means assuring whether the files for f_D and *Chip* are in the defined directory that were passed to f_D and *Chip* using the user defined function: *FileExists()* which is based on the built-in PHP function: file_exists() .

The transformed programs or products of $f_{D,}(\boldsymbol{\pi}_{\mathbf{D}})$ and $Chip(\boldsymbol{\pi}_{\boldsymbol{Chip}})$ form the transformed or target program ***P'',*** equation 3.5. More formally, to reflect the programs' references equation 3.5 is rewritten as equation 3.6. The equation 3.6 shows references from f_D to *Chip*, *Chip* to f_D and from f_C to *Chip*.

$$\boldsymbol{\pi}_{\mathrm{D}} = \mathcal{T}_{f'}\ (f_D) \tag{3.3}$$

$$\boldsymbol{\pi}_{Chip} = \mathcal{T}_{*}\ (Chip) \tag{3.4}$$

$$\boldsymbol{P''} = \boldsymbol{\pi}_{\mathrm{D}} + \boldsymbol{\pi}_{Chip} + f_C \tag{3.5}$$

The f_D is using all transformation options with the exception of the *function name* option as in equation 3.3. The *Chip* is also transformed but with all available options of the transformer, equations 3.4. The equations 3.3 and 3.4 produce equation 3.5 based on equation 3.2, after the transformations of f_D and f_C .

$$\boldsymbol{P''} = \boldsymbol{\pi}_{\mathrm{D}} \leftrightarrow \boldsymbol{\pi}_{Chip} \leftarrow \boldsymbol{f}_C \tag{3.6}$$

f_C *and* f_D stands for a set of function calls (*WorkSpaces*) $f_C = \{ f_{C1}, f_{C2}, f_{C3,\ldots}, f_{Cm}\}$ and $f_D = \{ f_{D1}, f_{D2}, f_{D3,\ldots}, f_{Dn} \}$ respectively, with both referencing or linked to the *Chip.*

where: ↔ **:** Transformation output produced ($\boldsymbol{\pi}$) on left and right reference each other

← **:** includes or references

= : "forms"

$\mathcal{T}_*$**:** Transform using all available options

π_x**:** product of transforming x

$\mathcal{T}_{f'}(f_D)$**:** Transform f_D using all transformation options with the exception of the "*f*" option, the *function name* obfuscation option

***P', P''*:** Modified versions (target program) of $\mathcal{P}$

3.8 *Chip* Anatomy and Control Flowchart

The *Chip* anatomy gives the skeletal overview of what the *Chip* should contain with some sample values of the variable parameters without putting into consideration the programming language used in the development.

```
ini_MAC_Address="##-##-##-##-##-##";
Software_ID="EL_ug_1_owner_id"; // Fingerprinting
bfilename="Function definitions directory";
Decision="Block";
Mode="Offline"; // or Online
function getSysMAC(){…}
function decision(){…}
function  MACAddressValidation(){…}
function isOnline(){…}
function sendRequest(){…}
```

```
Proper Function calls here
```

The *Chip* has three main segments: The parameter passing, function definitions and the function calls segments. In the first segment all values to be passed to the parameters are set and the necessary functions are defined in the second segment of the program in which the number of functions defined may vary from developer to developer, depending on the number of functionalities to be implemented. The last segment gives room for calling the defined functions in the second segment. The segment code gives the skeletal aspect of the *Chip* program.

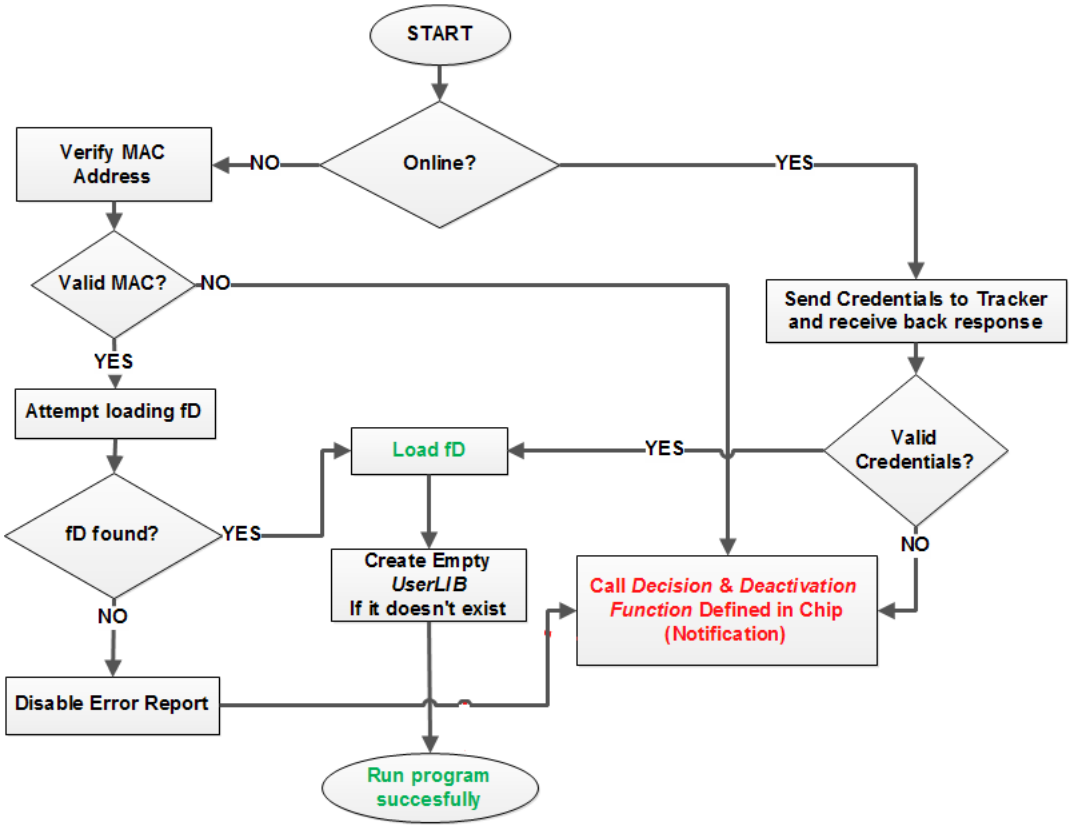

Fig. 3.3: *Chip* Work Flow Diagram (Anatomy)

The ***ini_MAC_Address*** variable stores the initial MAC Address of the user's PC that was registered on the server which will always be compared with MAC Address of the system the *Chip* runs on. The ***Software_ID*** stores the id of the Software which constitutes the user Id itself.

The ***decision*** variable stores the decision to be taken if the software is pirated in which the likely variable values may be: ***block, warn, warnAndBlock, Remove*** and so on. ***Bfilename*** stores the directory of the function definitions ($\boldsymbol{f_D}$) while ***mode*** holds the tracking status to be considered during the software tracking.

The *Chip* Anatomy in Fig. 3.3 gives the structure of the working principles of the developed *Chip* at the implementation level in Chapter four. The *Chip* was designed to first check the system online status. If the system is online the program tries sending credentials to the Tracker to verify the validity of the sent credentials and logs the details. If the credentials are valid it loads $\boldsymbol{f_D}$ and the program works properly else the deactivation function is called. Its response depends on its definition. But in our own case we only use a soft approach by displaying a notification.

Setting value of the online ***mode*** to "*Offline"* makes the *Chip* to check the system MAC address with the embedded MAC address in the *Chip*. If the MAC addresses are not the same the system display a notice and call the deactivation functions else the *Chip* attempts to load $\boldsymbol{f_D}$. In the case where f_D is first loaded, an empty *UserLib* is created, as no user library in the first run. Also if $\boldsymbol{f_D}$ is not found the error message setting is turned *Off* and the deactivation functions are called.

3.9 ER Diagram: Tracker Database

As mentioned at the beginning of this chapter that the tracker DBMS was implemented using MySQL, the ER Model of the database shown in Fig. 3.4 was designed using MySQL Workbench. The database constitutes of eight basic schemas. The table, *tblusers* stores the Tracker's administrator credentials with a unique *username*. The table *tbluser* serves as the "root" table, whereby the administrator writes records into: *tblversions, tblsoftware, tbldecision,*

tbldeveloper and tblownedby. A new application is registered based on its category, version, code and the decision to be taken when the application is tracked as pirated based on its predefined code, category and version. A developer is registered and mapped to an application with a unique fingerprint generated based on the code, category, code and the developer's ID. The schema, *tbldocumentation* holds the defined functions by a developer for a particular software project. The *tbltracks* schema stores the tracking history of all pirated and non-pirated software with their piracy status based on the decision defined in *tbldecision*.

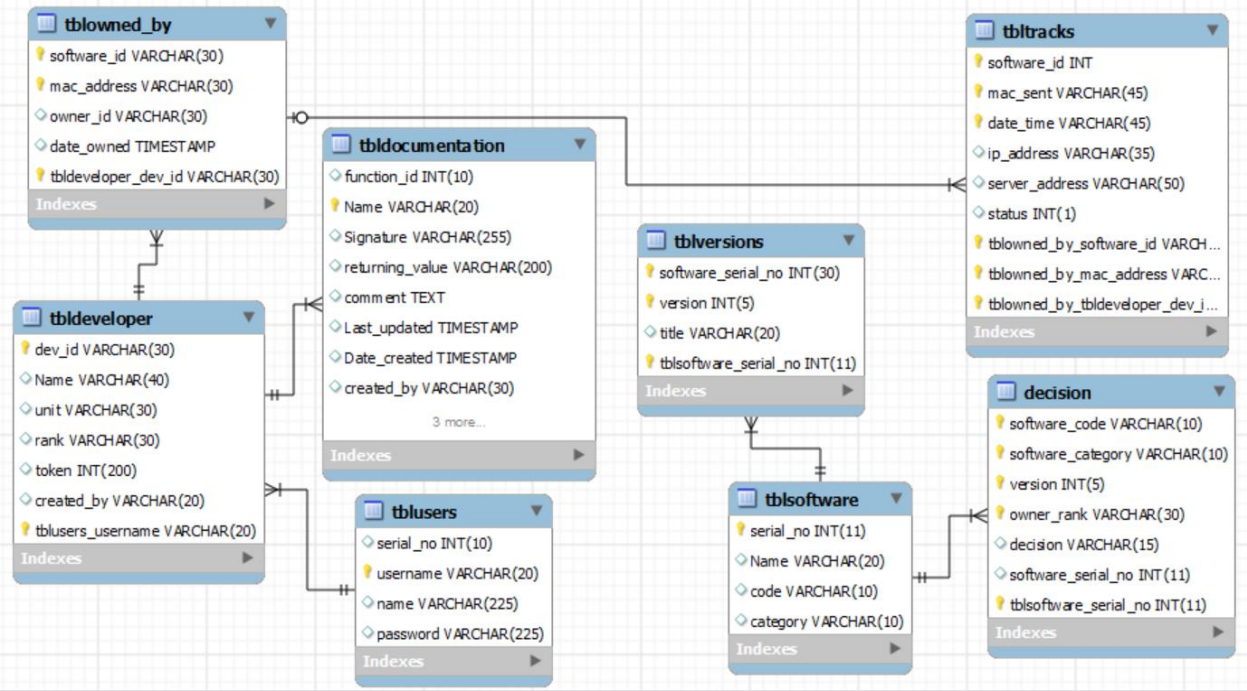

Fig. 3.4: ER MODEL for Tracker database

3.10 Programming Language and Database Used

PHP was used as a server-side scripting language during the implementation of the testing applications. This is because it is one of the commonly server-side scripting languages used in the software development (Clavijo, 2013). MySQL DBMS was used at the backend of the *Tracker* database as it is more prominently used in moderate and less complicated systems.

3.11 Obfuscation and File Encoding Software Used

The two obfuscators used in this research are: PHPLockIt!, which details can be gotten from the official website (PHPLockIt!, 2004). The obfuscator obfuscates and encodes PHP scripts with numerous options provided in it with some additional options for optimization. The second obfuscator is PHPObfuscator developed by Raizlabs. We combined the two obfuscators for better security (Aravalli, 2006). PHPObfuscator was used for the layer obfuscation (encryption) while PHPLockIt! was used for the file encoding and the remaining available options in it.

3.12 Testing Applications: Case I and Case II Scenarios

As of the derivation in Section 3.6.1, the definition of $\boldsymbol{f_D}$ for new (**Case I**) and existing (**Case II**) applications must differ because different programmers have different code organization and this makes it hard to generalize $\boldsymbol{f_D}$ as no defined way for collating functions for $\boldsymbol{f_D}$. The more functions extracted from the application to $\boldsymbol{f_D}$, the more secured is the application. With this there is a need to define a way of measuring weight of $\boldsymbol{f_D}$.

Assuming the universal set $\mathbf{U} = \{f_{D1}, f_{D2}, f_{D3}, f_{D4}, f_{D5}, f_{D6}, f_{D7}, f_{D6}, ..., f_{DL}\}$ as the set of files or sets containing all extractable functions in an application and $\boldsymbol{f_D} = \{f_{D1}, f_{D2}, f_{D3}, f_{D4}, f_{D5}, f_{D6}, f_{D7}, f_{D6...}, f_{DK}\}$ as the set of files containing function definitions. That is, for $\mathbf{U}$ and $\boldsymbol{f_D}$ we respectively get:

$$n = \sum_{i=1}^{L} n\,(f_{Di}) \quad \text{and} \quad p = \sum_{i=1}^{K} n\,(f_{Di})$$

Where p <= n and $n()$ returns the number of elements or functions in a set f_{Di} .

$\mathbf{U}$ is made up of function definitions based on the application logic and interface definition. Traditionally, it is obvious that the number of functions based on interface are less

than that of application logic, as interface are less extended compared to the logic in an application. Based on this we consider the number of application logic functions not to be less than $\lfloor n/2 \rfloor$ for our proposed approach.

The condition for CASE I holds when p > $\lfloor n/2 \rfloor$ for new applications designed based on our proposed approach and CASE II holds if 1 < p <= $\lfloor n/2 \rfloor$ for an existing application. Albeit Case II can be improved to fit the condition for Case I by successive function extraction as shown in Fig. 3.5.

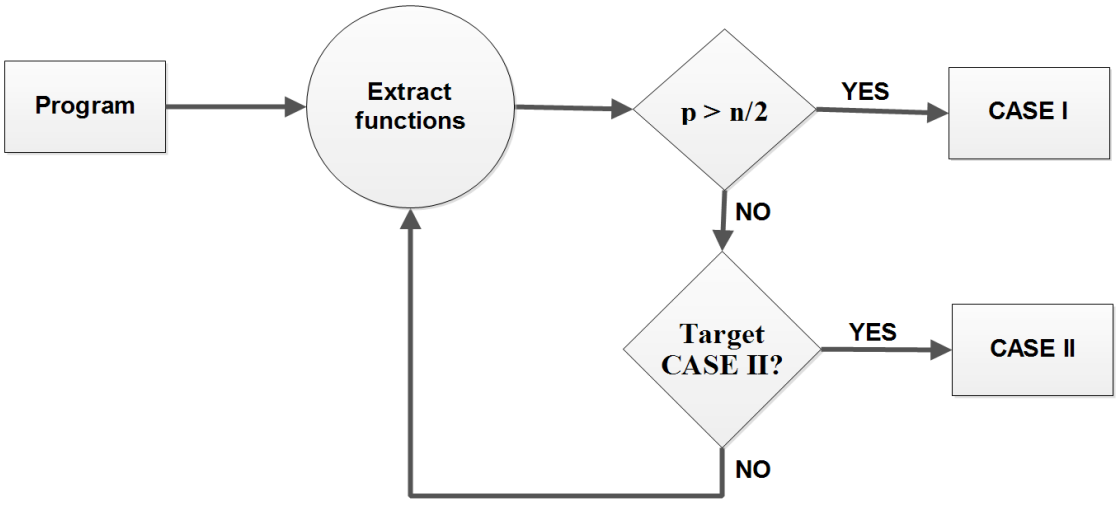

Fig. 3.5: CASE conversion and Function extraction Phases

3.12.1 ELogicPLUS

It is a software primarily meant to test the feasibility of our proposed approach. It is designed based on this research architecture. Mainly meant for Lecturers and course coordinators to reduce large results collation problem as results of numerous Continuous Assessments (CAs) are to be mapped to respective student's Exams result for a large class size, communicating students' results via SMS and unify lab score results and formatted results input for examiners' software consumption.

3.12.2 ExamsLOGIC 2.0

It is the official Examination Software developed and used by Ahmadu Bello University (ABU) since 2010. It helps in reducing overhead of examinations result data entry for examination officers and computing students' cumulative GPA (Grade Point Average) with dozens of reports and a lot of related results computations like Moderation, Results Statistics and so on. The software synchronizes data between the Student and Staff portals. There is a challenge on how to protect the code of ExamsLOGIC 2.0 using our proposed approach as it is an existing Software.

The two software are standalone although some of the tasks are based on synchronization and SMS sending for online tasks.

3.12.3 Why the Applications?

These applications were used for testing because of the following reasons:

1. Easy access to the full source code of the applications
2. Partially dependent on online functionalities attached to the applications which facilitate the online tracking

3.13 Steps to Implement the Architecture

The flowchart in Fig. 3.6 shows the procedure to develop the components of the proposed approach starting from *Chip* though the developer can begin with any of the three components. Developing the *Chip* simply means writing the *Chip* code which depends on the programming language used. The *Chip* can then be obfuscated with all obfuscation options. If it is a new application (Case I), new or major functions of the applications are extracted and collated as $\boldsymbol{f_D}$ after adding and calling the *secure function*. But if it is an existing application (Case II), existing functions are collated and if there are new functions that can be introduced based on the common

functionalities and interfaces they should be added to the function collator until the abstraction is sufficient, P> $\lfloor n/2 \rfloor$. Then we save it as $\boldsymbol{f_D}$ just as in Case I. All necessary parameters are passed to the f_D and then obfuscate it without obscuring the functions' name. Finally install *Chip*, $\boldsymbol{f_D}$ and $\boldsymbol{f_C}$ on the user's PC.

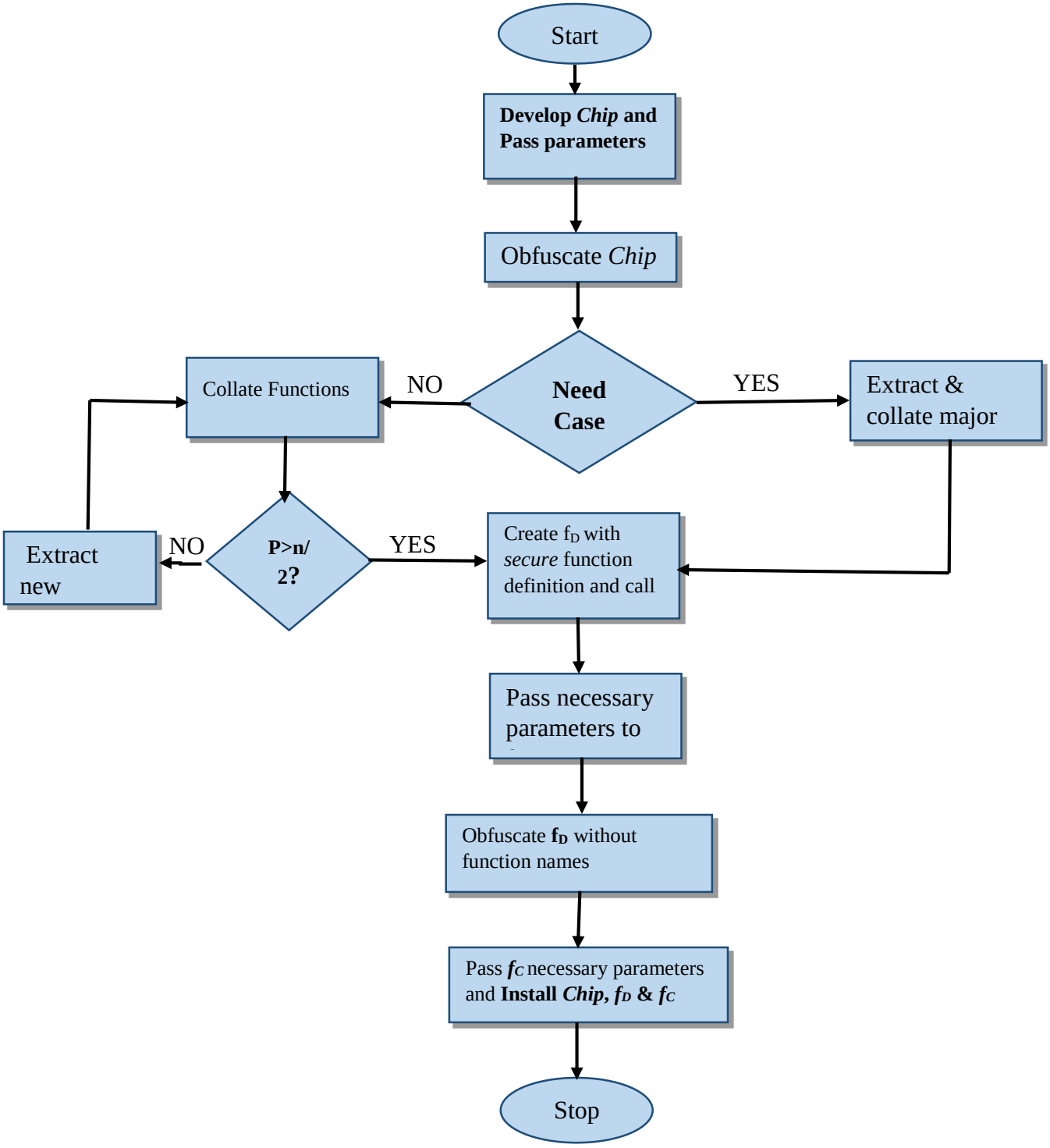

Fig. 3.6 : Flow chart of Architectural Implementation

To implement the proposed approach the steps below for both *Case I* and *Case II* should be followed:

Step 1:

i. Develop the *Chip* based on the proposed *Chip* anatomy in Fig. 3.3 using a programming language of your choice

ii. Pass the *Chip* parameters itemized in 3.9.1

Step 2: Create an empty function definitions ($\boldsymbol{f_D}$) file consisting of a definition and a call of the secure function which consists of the relative *Chip* path or directory.

Case I: Successive abstraction of the common task into function should be done and save the resulting functions in $\boldsymbol{f_D}$ file.

Case II: Collate all existing functions into a single $\boldsymbol{f_D}$ file and reapply function abstraction if possible. Here we assumed most of the functions are placed in a *function* file of an existing Software.

Step 3: Transform $\boldsymbol{f_D}$ with the exception of the ***function name*** obfuscation parameter. Using an obfuscator of your choice.

Step 4: Transform *Chip* using all the obfuscator's options

Step 5: Include or reference the *Chip* and *UserLib* in all functions' call ($\boldsymbol{f_D}$) files (*Workspaces*)

Also the developer should continuously update his *UserLib* and *UserDoc* simultaneously by placing new functions into his *UserLib* and their documentions based on the function names, parameters and the respective values they return.

3.14 Bulk *Chip* Production

The pseudo code below shows the production of the *Chip* for many users in a company mapping each copy to a particular user in a company, using the fingerprinting approach. This should probably take place on the Tracking server as records of the users with their respective system MAC addresses are available on the server. The obfuscation algorithm must be residing on the server to generate an obfuscated *Chip* else only set of non-Obfuscated *Chip* files will be generated.

n=1
for user-n
{
file$_{\text{user-n}}$=**create_file**(user_ID,Software_ID,MAC_Address,Decision, Mode,f_D path,Chip_code) ;

$\boldsymbol{\pi}$***chip***$_{\text{user-n}}$ = $\mathcal{T}_*$ (**file**$_{\text{user-n}}$);

save($\boldsymbol{\pi}$***chip***$_{\text{user-n}}$);

email($\boldsymbol{\pi}$***chip***$_{\text{user-n}}$);
n++; // next user
}

3.15 Assumptions Made

We assume only the administrators have access to untransformed source codes of *Chip*, $\boldsymbol{f_D}$, and the splitting rule that results to the split components of the initial software source code. Synchronization of files between the clients and the central repository which in this case is the Tracker, is assumed to be done using any project management system like the *Mercurial* repository with *Bitbucket* cloud mentioned earlier.

CHAPTER FOUR:

System Implementation, Testing and Evaluation

4.1 Introduction

This chapter consists of the procedure to implement the proposed system, explanation of the actual derivation used in the implementation of the system, testing the efficacy of the system in both CASE I and II using the two software as the testing platforms, explanation of some segment code as used in the implementation, comparing the proposed architecture with existing work.

4.2 Specific Derivation

A good obfuscation composes of one or more code transformations that transform a program's control and data flow in order to make reverse engineering harder. The only restriction for the transformations is preserving the semantic meaning of the target program. Combining more than one transformation sequentially on a piece of code is possible at code compilation level (Cappaert, 2012). Also, using more than one or same transformer many times makes the code harder to be easily reverse engineered (Aravalli, 2006).

Since we are using two obfuscators, *PHP Obfuscator* ($\mathcal{T}'$) and *PHPLockIt!* ($\mathcal{T}''$) in the actual implementation, in order to make the encryption more secured as mentioned in Section 3.6. Equations 3.3 and 3.4 will transform into specific equations 4.1 and 4.2 respectively.

$$\pi_D = \mathcal{T}''_{e}(\mathcal{T}'_{f'}(f_D)) \tag{4.1}$$

$$\pi_{Chip} = \mathcal{T}''_{*}(\mathcal{T}'_{*}(Chip)) \tag{4.2}$$

$$\boldsymbol{P}' = \mathcal{T}''_{e}(\mathcal{T}'_{f'}(f_D)) + \mathcal{T}''_{*}(\mathcal{T}'_{*}(Chip)) + f_C \tag{4.3}$$

The *Chip* reference to f_D was done within the load function definition as shown in Fig. 4.1 so it checks for f_D before attempting to load it.

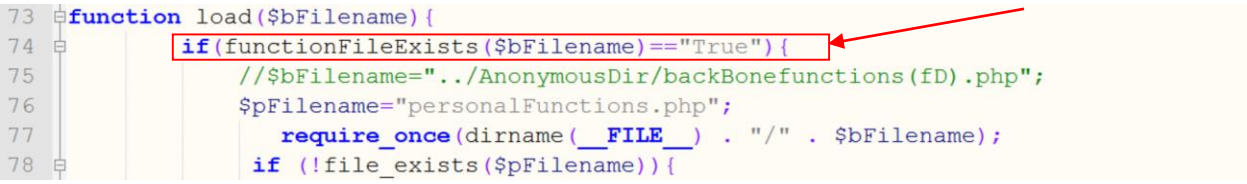

Fig. 4.1: Chip referencing f_D

```
function secured(){
     $bFilename="Chip.php";// Specify Chip directory
     if (!file_exists($bFilename)){
     echo "Please, get a clean copy.";
          die(); //Script from running
     }
}
secured(); // calling the function within fD
```

The above code segment is the definition of the `secured ()` in f_D. It first checks if the *Chip* is on the specified location before it allows the f_D to be run. It prints a notice and stop the script from running the remaining functions defined in f_D. It works that way because the `secured()` definition and call are placed at the beginning of f_D. And if it is successful it loads the remaining functions definition in f_D.

The database string connection file (*db.php*) is also an element of the set f_C, hence it loads the *Chip* using the segment code:

The element (*db.php*) of the set f_C is loaded using the segment code:

```
$chipDir="Chip.php";// To be Added

require_once(dirname(__FILE__) . "/" . $chipDir);//To Reference
```

This is because *db.php* is used by almost all elements in f_C. The programming workspace comes immediately after the segment code in the f_C elements.

```
3  $dbDir="db.php";
4  require_once(dirname(__FILE__) . "/" . $dbDir);//
5  ////////////////////////////////////////
6
7  //Programmer's code (Function Calls here)
```

Fig. 4.2: f_C with Developers' Workspace

4.3 System Testing for the Two Scenarios

After implementing the proposed architecture using the two testing applications (ExamsLOGIC and ElogicPLUS) Fig. 4.3 and Fig. 4.4 shows a snapshot of the applications tracking an unregistered users' PCs and displaying a notification. Though this depends on the users decision on what to do if an application is claimed to be pirated based on the business rules defined in the *Chip* logic as explained in Chapter Three.

The Applications attempted to load their respective ***db.php*** files which were linked to the *Chip* that does the Checking. The *Chip* validates the PCs' MAC addresses by testing/comparing the MAC address that was passed to *Chip* with the current system MAC address on which the application runs on, and takes decision based on the passed ***decision*** values. This happens when ***mode*** is set to "Offline", but if is set to "Online" it sends the fingerprint which contains the software Code, Category, version and the user ID. Besides, IP address and current system MAC

address are also send to the Tracker to save the trace record and its validity. It then sends back a response based on the decision value attached to the version of the application on the Tracker.

On the tracker we assigned privileges based on application name, version, user's level and decision to be drawn when a particular version of the application is pirated and tracked. The details are gotten from the sent fingerprint within the *Chip*.

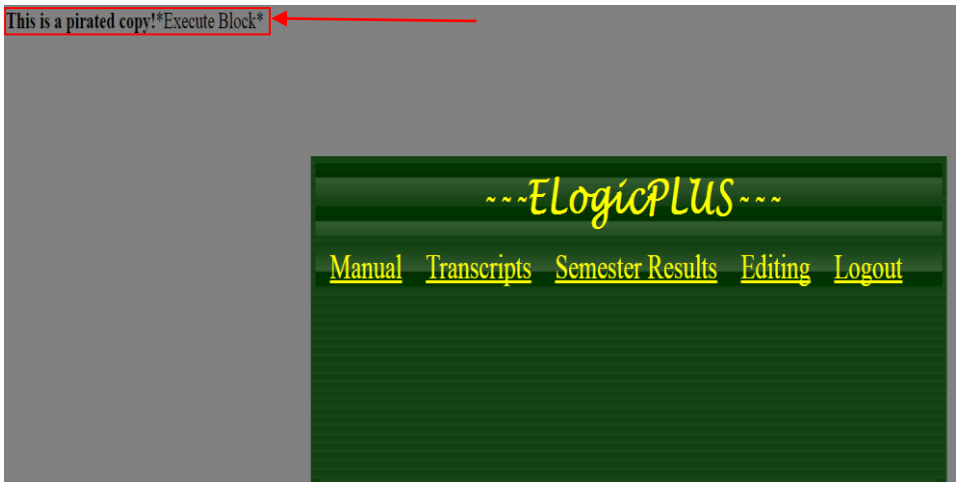

Fig. 4.3: Testing Architecture with CASE I

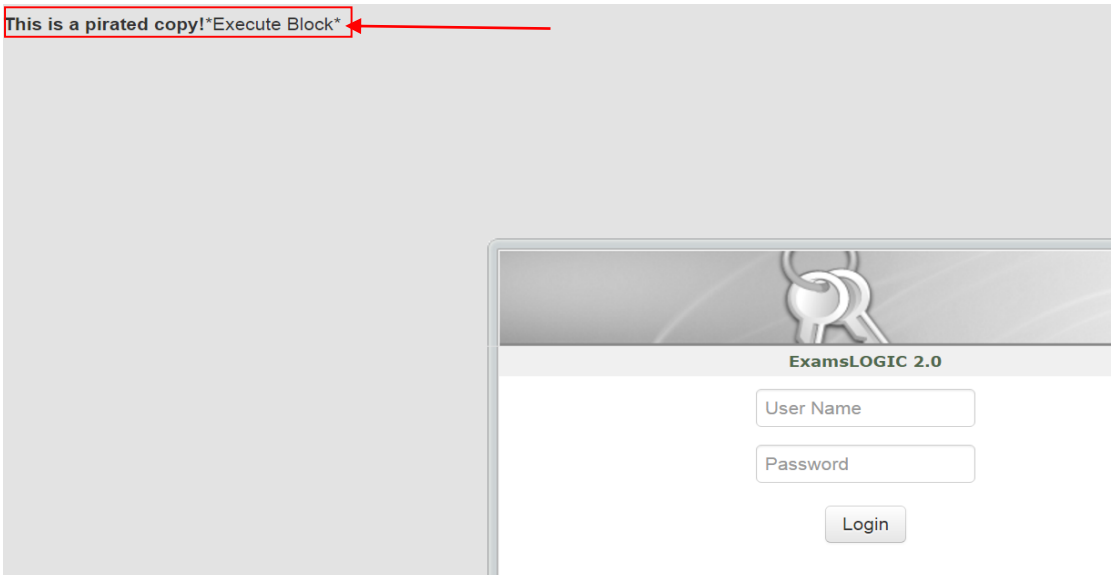

Fig. 4.4: Testing Architecture with CASE II

4.4 Research Results Analysis

This section provides comparison between the previous research work of Das (2014) and the proposed system based on the code splitting technique used. Table 4.1 gives the comparison of the results.

Table 4.1 Result Comparison

Research Work	Environment		Clear Splitting Rules	Hardware Aspect		Static	Dynamic	Tracking
	Development	Production		Software	Hardware			
Das (2014)	✗	✓	✗	✓	✗	✓	✓	✗
Proposed System	✓	✗	✓	✓	✓	✓	✗	✓

4.4.1 Result Discussion

Das (2014) work focused on controlling both static and dynamic reverse engineering taking advantage of some of the operating system functionalities, the software component. While in the proposed approach we focused only on the static approach because of the type of obfuscators we used in the implementation. We put into consideration both the hardware and software aspects of the system. Das (2014) applied code splitting approach but a clear rule on how to split the target program code was not mentioned as it was left opened though the major split parts (*Server* and *Client*) were mentioned. Also, his work was based on the production stage as it was applied on a target program while in the proposed system we are concentrating on a source program, at development stage. Table 4.1 shows the comparison parameters.

CHAPTER FIVE

Summary, Conclusion and Recommendations

5.1 Introduction

This chapter provides a summary of the research work that encompasses the motivation, statement of the problem, methodology, results, conclusions drawn from the result of the research and recommendations for future work.

5.2 Summary

Software companies have been facing challenges pertaining to protecting their products from adversary's attacks. This urged the companies to devise legal and technical means like watermarking, tamper proofing, fingerprinting and obfuscation approaches to curb piracy. Obfuscation is one of the technical approaches primarily developed to control reverse engineering of software products at production stage.

In this research work we proposed and implemented an architecture to curb software piracy at development stage using code-splitting, fingerprinting and obfuscation techniques. The system was able to track pirated software in online or offline modes. The feasibility and effectiveness of the approach was tested. We compared based on the environment each approach was applied, clarity of the splitting rule, aspects pertaining the hardware and software of the system, type of prevention attack(s) handled: Static or dynamic and the tracking capability of the systems.

5.3 Conclusion

Although obfuscation techniques were primarily meant for use at production stage, we can conclude based on the result of the proposed approach that it is feasible to also employ some standard protection techniques to control piracy of software instances at development stage.

5.4 Recommendations

Many researches have been done on controlling static and dynamic analyses or reverse engineering at production stage. With this we recommend using some existing approaches that tackle both static and dynamic reverse engineering to measure the efficacy at development stage.

Also, research on MAC address protection (hardware aspects mapping) should be explored and, if possible, an alternatively more secured hardware approach should be used. Location or co-ordinates of the user's PC should be part of the details sent to the tracker which may give at least a rough location.

We believe research in the field of software security at development stage is crucial and still in its infancy. Hence, more techniques can be developed as improvements or alternatives to our proposed technique.

References

AliMunassar, N. M., and Govardhan, A. (2010). A Comparison Between Five Models Of Software Engineering. *International Journal of Computer Science Issues, IJCSI, Vol. 7*, P. 94-101.

Aravalli, S. (2006). *Some Novice methods for Software Protection with Obfuscation.* University of New Orleans Theses and Dissertations.

Asongu, S. A., and Andrés, A. R. (2012). Fighting software piracy: which governance tools matter in Africa? *African Governance and Development Institute*.

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., and Yang, K. (2001). *On the (Im)possibility of Obfuscating Programs.* Technical report, Electronic Colloquium on Computational Complexity.

BSA. (2007). *Fifth Annual BSA and IDC Global Software*. Business Software Alliance retrieved on May 27,2014 from http://globalstudy.bsa.org/2007/studies/2006globalpiracystudy-en.pdf.

BSA. (2011). *Software Piracy in the EU: BSA Response to the Commission Report.* Business Software Alliance retrieved on May 27,2014 from http://globalstudy.bsa.org/2011/&usg=AFQjCNE5IbLfJ_sTr-JK.

BSA. (2010). *Business Software Alliance.* Global Piracy Study.

Cappaert, J. (2012). *Code Obfuscation Techniques.* Belgium: Katholieke Universiteit Leuven – Faculty of Engineering Arenbergkasteel, B-3001 Heverlee.

Clavijo, D. (2013). *Server-side programming language statistics.* Retrieved on May, 9th 2015 from http://blog.websitesframeworks.com/2013/03/programming-language-statistics-in-server-side-161/.

Collberg, C., and Thomborson, C. (2002). Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.

Collberg, C., Thomborson, C., and Low, D. (1997). A Taxonomy of Obfuscating Transformations. *Technica lReport#148, Department of Computer Science, The University of Auckland, Private Bag92019, Auckland, New Zealand*.

Collberg, C., Thomborson, C., and Low, D. (2000). *Breaking Abstractions and Unstructuring Data Structures.* The University of Auckland: IEEE Xplore.

Das, S. S. (2014). *Code Obfuscation using Code Splitting with Self-modifying Code.* Odisha: NIT Rourkela.

Fischer, J. A., and Andrés, A. R. (2005). Is Software Piracy a Middle Class Crime? Investigating the InequalityPiracy Channel.

Janssen, M. (2016). *Combining Learning with Fuzzing for Software Deobfuscation.* Delft University of Technology http://repository.tudelft.nl/.

Lewallen, R. (2005). *Software Development Life Cycle Models.* Retreived on May,2015 from http://codebetter.com/raymondlewallen/2005/07/13/software-development-life-cycle-models/.

Myles, G. M. (2006). *Software Theft Detection through Program Detection.* Department of Computer Science, The University of Arizona.

PHPLockIt! (2004). *[Computer Software]*. Retrieved on May, 20 2014 from http:// www.phplockit.com.

Richa. (2014). *Reverse Engineering Tutorial: How to Reverse Engineer Any Software.* Retrieved on March, 2016 from https://blog.udemy.com/reverse-engineering-tutorial/: udemy.

Rouse, M. (2005). *Piracy.* Retrieved on May, 2015 from http://whatis.techtarget.com/definition/piracy.

SDLC. ((n.d.)). *Waterfall Model.* Retrieved on May, 21 2015 from https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm.