Software Piracy Protection and Detection using HMM, Code Obfuscation, Dynamic Code Mutation, and cryptographic techniques

[1]Sakshi Ganesh Dighe, [2]Kshitij Kumar, [3]Harshit Jain, [4]Samyak Jain, [5]Dhairya Jain

[1]Associate Engineer, Danfoss,Pune,India
[2] Systems Engineer, Infosys, Bangalore, India
[3]Assistant Product Manager, Sawingz, Bangalore, India
[4]Software Engineer, CHOYS Technologies
[5]Undergraduate Student, (Computer Science Engineering), ADGIPS, Delhi, India

***Abstract*- In the contemporary era characterized by pervasive internet usage, the escalation of software piracy poses a substantial challenge, exacerbated by advancements in computer technologies. In response to these malicious incursions, the integration of code obfuscation techniques presents a viable strategy by transforming programs into resistant patterns. This research introduces a novel obfuscation methodology that amalgamates nontrivial code replicas with established obfuscation practices, tailored to meet specific efficacy requirements. The justification for this approach stems from the imperative need to address prevalent security risks. The proposed methodology is elucidated, accompanied by an illustrative example. Furthermore, the study advocates for a software piracy prevention mechanism, wherein established cryptographic techniques including Triple DES, Zero-knowledge proof, and enhanced RSA are employed to fortify the system. The devised system incorporates a projected technique for safeguarding software documentation and records, coupled with a proposed methodology for generating a unique software Copy Identification Number (ICN). Through the analysis of opcode sequences derived from these altered replicas, a proficient model is employed to identify suspicious software and ascertain its resemblance to the original program. A higher score indicates a high likelihood that the suspicious program represents a modified version of the underlying software. Additionally, dynamic code mutation is introduced as a countermeasure to complicate reverse engineering efforts. Conversely, a lower score suggests substantial dissimilarity between the suspicious software and the grounded software. This research demonstrates the resilience of the proposed technique, emphasizing that significant modifications to the underlying program are necessary before its identification becomes feasible.**

***Keywords*- Improved RSA, Triple DES, Zero-knowledge proof, Hidden Markov models, code obfuscation, Code Encryption, Dynamic code mutation.**

I.INTRODUCTION

Software security has become one of the most enticing topics with a great financial interest, luring everyone from enormous software vendors to content producers such as the film & audio recording industries. Software's digital data is extremely susceptible. Data authenticity and confidentiality are two key security concepts. In the context of software systems, the process of reverse engineering, wherein an individual seeks to acquire a comprehensive or partial comprehension of the higher-level structure of an executable program, poses a significant concern. Malicious actors can exploit recent advancements in program analysis technology and software engineering tools to undermine existing safeguards. Consequently, such technological capabilities empower attackers to identify software vulnerabilities, execute unauthorized modifications such as bypassing password protection, and manipulate the program by identifying and removing copyright notices or watermarks. Furthermore, this facilitates the illicit extraction of intellectual property, amplifying the potential for unauthorized access and exploitation of proprietary information. Data concealment ensures a message's data privacy, whereas data authenticity ensures the message's integrity. Software protection lies within the purview of several disciplines, including security, cryptography [17], and engineering. Among the most challenging concerns for s/ware suppliers is securing code against assaults such as reverse engineering [18], analysis, and manipulation. If a rival is successful in getting and utilizing an algorithm, it will cause a huge problem. Further, it is not intended that security-related code, sensitive data, or secret keys should not be devoured, obtained, pillaged, or annihilated. Despite legal safeguards like trademarking & cybercrime regulations in position, these approaches continue to pose a significant risk to privacy advocates & software developers. Forms of Software Piracy The Business Software Association [BSA] classifies software piracy into five categories [1]:

1) The end-user infringement happens when an end-user replicates s/ware outside permission. It might express itself in either of the subsequent forms: a) An end-user attains a solitary licensed replica of the program & installs the same on many machines. b) The software installation DVDs are replicated and distributed. c) An end-user buys & establishes an advancement without having beforehand purchased an authorized version.
2) The client-server model infringement happens whenever a program is set up on a networked computer & is being utilized via numerous individuals compared to those licensed to use it.
3) Cyberspace theft happens whenever illicit replicas of software are accessible for free or for a price on the Internet.
4) When unauthorized software is mounted on a novel workstation & vented, hard-disk loading happens. This practice frequently occurs whenever a corporation is endeavoring to diminish expenditures to formulate its products considerably alluring.

In this research contribution, a novel technique is presented for the automatic exacerbation and/or introduction of challenges within existing software programs. The core concept revolves around the dynamic mutation of a program during its execution, causing a designated memory region to host various code sequences throughout its runtime. The implications of this technique are explored, particularly its capacity to disrupt assumptions ingrained in prevalent analyses utilized for reverse engineering processes.

Software-Based Security: Software-based security approaches rely on the same distributed software. Leveraging s/ware as a safeguard paradigm offers several advantages, notably more dissemination mobility and decreased protection added cost.

Collberg et al. [19] define 'code obfuscation' as a family of procedures that turn a "basis program P into a direct program P' so that P & P' have the same "observable behavior" & P' is intricate for an attacker to converse engineer". According to [19], the following requirements must be met for an obfuscating alteration from P to P' to be a lawful obscuring makeover: [If P fails to terminate using a fault ailment, P' might or might not expire], [Or else, P' necessarily terminates & P' must generate the identical o/p as P].

Encryption: The goal is to encode a disseminated program and require a ['decryption key'] to execute it. Countless 'encryption approaches', like having several 'encryption keys', could be utilized [2]. Content security approaches count on cryptographical procedures in which 'the decryption key' should be kept obscured from (dishonest) users. In this study, a suggested software protection approach based on cryptographic algorithms is implemented [3].

Cryptography: The act of generating an encrypted output, known as ciphertext, by combining certain input data, known as plaintext, with a user-specified key so an arrangement that nobody could reasonably decipher the plaintext sans the key that is used for encryption. Ciphers are the algorithms that mix the keys and texts [4].

Symmetric Cryptography System: A transmission of info. Is decrypted & encrypted via a single 'secret key', as in traditional symmetric encryption. The most popular kind of symmetric encryption (DES) is known as the "Data Encryption Standard." Despite being replaced by the Advance Encryption Standard (AES), DES is still the most important encryption method. (However, the DES designation has been revoked). DES (particularly Triple-DES) remains immensely popular despite its removal [5].

Triple DES (TDES): It is a *block cipher* constructed by augmenting the 'Data Encryption Standard (DES)' cipher 3 epochs. When it was determined that a "56-bit" DES key was inadequate to fend against "brute forces attacks," the TDES was chosen as an easy way to increase the "key field" without switching to a novel method. To stop "meet-in-the-middle attacks," which successfully circumvent "double DES encryption," three stages are necessary.

TDES's most basic variation works as comprehends: "DES (k3; DES (k2; DES (k1; M)), where M is the encrypted message block and k1, k2, and k3 are DES keys" [6]. Triple-DES is DES repeated 3 times with 2 keys utilized in a certain directive. (Triple-DES could also be performed via 3 distinct keys rather than just 2. In any scenario, the final 'keyspace' is around 112 [7].

Asymmetric Cryptography System: The public-key approach encrypts with one key & and decrypts with a separate but associated key [8]. RSA is the most popular of the public key algorithms. An enhanced RSA is investigated in this paper [9]. "The RSA scheme is a block cipher in which the original message and cipher message are integer values in the interval ..0[n −]1 where 'n' is a composite modulus." The novel message and cipher message from the overall linear cluster of (h h) matrices over 'n z' indicated by () n zhg and the novel message directed by m are both included in the suggested framework. The message is encoded in blocks using the RSA arrangement and then rifted to blocks; each block's value must change to be somewhat less than the modulus n. In other words, finding eth roots mod a composite modulus is the RSA difficulty. The requirements established by the modulus n and the public key e are intended to ensure that there is only one m for every integer c where m c n e = mod for every integer c.

Hidden Markov Models: A Markov process is a sort of 'statistical model' with recognized transition probabilities & states (Stamp, 2004). The states of a 'Markov process' are observable to the viewer.

II.LITERATURE REVIEW

Collberg et al. [10] presented a concise overview of the techniques for mitigating these concerns. It typically embeds secret, unique info. hooked on a program in a custom that it can be assured that a convinced s/ware occurrence belongs to a specific discrete or organization. Unless the watermark is crushed, this data can be used to distinguish duplicated software from the original.

Table 1: Existing Piracy Protection Approaches

Existing Approaches	Functionalities
[10] Collberg et al.	Breakdown of the techniques taken to combat these dangers. S/ware watermarking, however, for example, emphasizes rapidly securing applications from infringement.
[13] Hongxia Jin et al.	The emphasis is on identifying the attacker & conducting forensic examinations. The author presented a preventive surveillance procedure for combating a continual assault before collaboration.
[11] Cappaert et al.	A fractional encrypting solution based on code encryption was introduced.
[12] Chang et al.	An integrated ensemble of application defenders that constantly evaluate their respective coherence alongside that of the application's vital parts comprises the backbone of the developer's privacy methodology.
[16] Horne et al.	Self-checking code is used to avoid software manipulation.
[15] Song-kyoo Kim	Focuses on unpredictable preservation for s/ware protection employing an enclosed scheduling approach with suspicious records.
[14] Jung et al.	A key chain-based code block encryption technique was presented to safeguard software.

This study focuses on the security of a software program and the material it safeguards. Each year, the industries spend billions of dollars, mostly on software infringement. The capacity to safeguard software programs counter to modification and identify the assailants who release infringed replicas underpins the accomplishment of content/software security in a large section. Hongxia Jin et al. [13] focus on assailant documentation and legal investigation in this study. The emphasis is on identifying the attacker & and conducting forensic examinations. The author presented a preventive surveillance procedure for combating a continual assault before collaboration.

Chang et al. [12] presented a method grounded on s/ware fortifications. The author's security strategy is primarily grounded on an amalgamated system of s/ware sentinels that reciprocally check apiece reliability. An integrated ensemble of application defenders that constantly evaluate their respective coherence alongside that of the application's vital parts comprises the backbone of the developer's privacy methodology.

Cappaert et al. [11] proposed a partial encryption technique based on code encryption [12]. Users decipher the encrypted binary codes during runtime. Henceforth, A fractional encrypting solution based on code encryption was introduced.

Jung et al. [14] proposed a key string-based block-code cryptography approach for protecting software. In Jung's method, the fundamental block, which is component-size, is substituted with a component that interferes alongside and is a solved-size block. Cappaert and Jung both employ similar tactics. Jung's approach makes an effort to correct Cappaert's procedure's flaws.

Song-kyoo Kim [15] discusses the strategy to demonstrate the theoretical software protection approach. If the software components are recognized as alternatives in the specified architectural framework, the system's vulnerabilities might be subverted by adopting an unpredictable maintenance model that includes fundamental reliability with randomized supplementary reserves and substitution techniques.

A technique reminiscent of the runtime code generation approach proposed in this study was introduced by Debray and Evans [26] in the context of profile-guided code compression. Their method aimed to diminish the memory footprint of applications by storing infrequently executed code in a compressed format, only decompressing it when execution is necessary. Notably, this approach ensured that only a small fraction of the infrequently executed code was decompressed at any given point. However, due to substantial decompression overhead, the frequently executed code remained consistently available in its decompressed, original form. Consequently, this compression technique did not effectively conceal the frequently executed segments of a program, which, incidentally, are more likely to encompass the code warranting protection.

Horne et al. [16] described s/ware altering protection using *self-checking code*. Testers are pieces of code that check the integrity of code segments. It is feasible to take advantage of a unidirectional hashing algorithm with a predetermined hash value. The necessary action will have to be implemented to comply with the fidelity of the norm if it fails to be fulfilled. As the assortment of inspectors expands, the assailants grow increasingly perplexed, thereby making it impossible to convince them to penetrate the individuals who are testing.

III.METHODOLOGY

The approach employs a checksum corresponding to regardless of application it has been combined therewith. An encrypted value is produced every time the application goes online and the result is then contrasted to the hashed result which has been formerly preserved. If both variables match one another, the application executes; if not it collapses. Aside from this capability, it also changes every serial number character to hexadecimal or mangled data, making it extremely difficult to determine the serial number required to access the software. Figure 1 depicts the model's architecture. It is made up of the following sections.

Employing Nonlinear Measurements, Collberg et al. [19] convey a trio of metrics to assess the efficacy of code obfuscation strategies: Cost, Resilience, and Potency.

A. Principles linked s/ware intricacy measurements like McCabe's cyclomatic intricacy [20].

B. Grounded on Confrontation to Outbreaks,

Static Analysis Attack: In this kind of assault, the assailant constructs a CFG (Control Flow Graph, an elevated visualization of the instructions) of s/ware that aids in figuring out how the application performs. It can be found by empirically exploring programs [21].

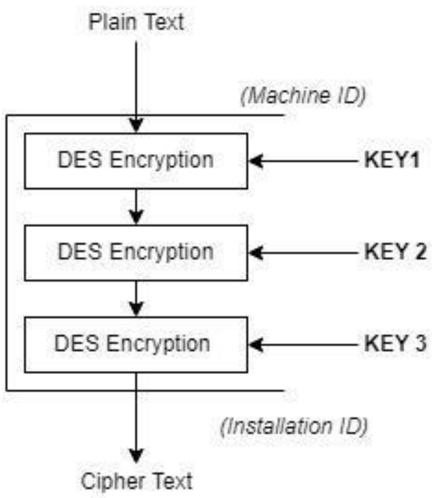

Fig 1: ID generation using TDES

An obfuscation strategy that upsurges an invader's analysis exertions is resistant to the attack. As explained by Schrittwieser et al. [23], the more analysis work required, the greater the resistance against reverse engineering assaults.

Dynamic Analysis Attack: An attacker conducts this attack by running the software on many inputs and inspecting the execution traces [21]. Dynamic analysis is more challenging than static analysis because the software must be performed with different inputs.

Cipher Replica Recognition Attack: An aggressor novelty & and eliminates code clones in a program to minimize the program's code size (Baxter et al. [22]).

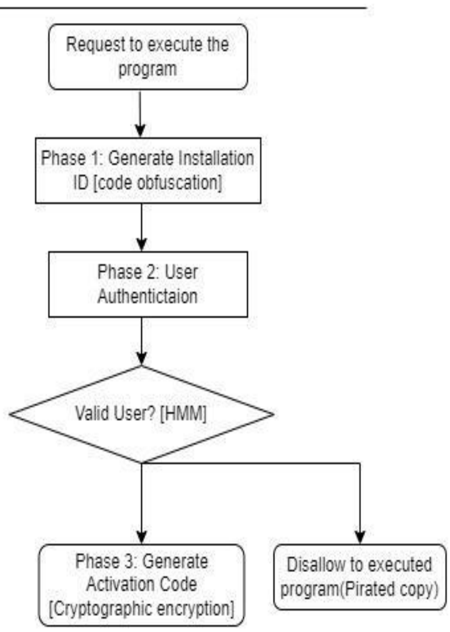

Fig 2: Proposed Design

Plain code in this study was developed in C++, which is straightforward to grasp even for noob programmers.

Encryption Tool: This is the program that we use to encrypt the software. The encryption tool encrypts the plain code, making it more difficult for a cracker to decipher.

Encrypted Code: Normal code is converted to encrypted code that is difficult to understand even for a programming expert using our encryption tool. Only when the correct key is used to decode the encryption can the encrypted code be read and understood. Encrypted software is notoriously tough for crackers to decipher. When a cracker struggles to grasp encrypted software, it suggests that breaking such a code is exceedingly unlikely.

Decoding Module: Using the decrypted key, the encrypted code is converted to a more readable format in this module. The decoded version of the code is now returned to its original basic form, making it easier to understand and read.

Finally, to produce our application licenses, the '*serial number generation'* is disguised via cryptography. It is a code transformation approach that preserves the functionality of the code but is twisted in such a manner that a software cracker cannot easily comprehend it. A serial key must be entered into the software via the interactive section on the first installation.

Dynamic code mutation: The clustering process is executed using a node-merging algorithm applied to a fully connected undirected weighted graph. In this graph, each vertex corresponds to a cluster of procedures, and the weight of an edge (A, B) signifies an estimate of the additional run-time overhead, i.e., the cost of edits, incurred when clusters A and B are merged. The determination of run-time edits for a cluster quantified as the number of control flow transfers between two members within that cluster, relies on profiling information obtained from a set of training inputs. This approach facilitates an estimation of the operational modifications required by a cluster.

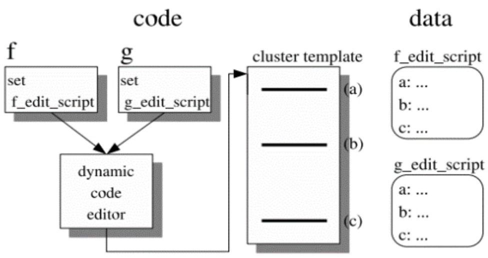

Fig. 3. Run-time code mutation with clustered procedures

As is customary, the clustering process involves navigating a performance trade-off. On one hand, there is a preference for maximizing the size of each cluster, aiming for larger individual clusters to reduce the overall number of clusters. This strategy enhances obfuscation by increasing the diversity of instructions mapped to the same addresses, thereby deviating from the conventional one-to-one mapping of instructions and memory addresses.

If the legitimate key is input, the inclined provides you with contact and you might perform it. The game will not allow access "if the serial number entered is invalid." The serial number code portion was built, and a keygen.exe executable file was created. As soon as 'double-clicking', 'the keygen.exe', it immediately creates the 'serial number'.

This paper focuses on securing delicate elements of software, like licensing verification & and 'data masking' methods, in this study. Delicate components are often a minor fraction of the overall software in many software applications. Such components are typically a few hundred lines of code in our internal projects.

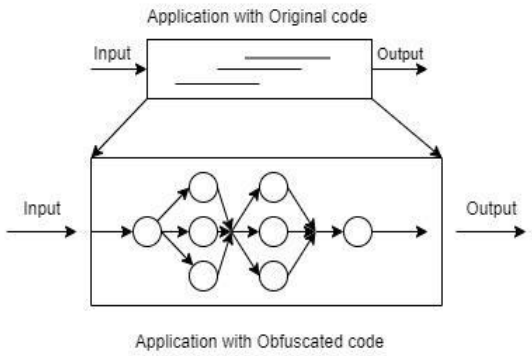

Fig 4: Code Obfuscation Scheme

Given such sensitive software components, we offer a 4-step technique to obscure them. Split the subtle bits into rational code pieces first. Finally, using dynamic predicate variables, connect the clones to the matching original pieces to generate legitimate control flow pathways, 1 of which is arbitrarily picked at run-time. Figure 4 depicts this process, with ellipses representing logical code clones built for obfuscation.

Step 1: Rational Cipher Wreckages because of instinctive technique for identifying subtle code fragments, to provide sensitive code snippets. With this requirement, we separate the program providing elusive functionality to rational cipher parts.

Step 2: Complex Code Clones: This protects against static analysis attacks at this level. A semantically equivalent code specimen C' is a clone of code fragment C. In other words, replacing "C in a program P with C'" results in a program "P' that produces the same output as P for all inputs (unless P terminates for that input)". Article [19] discusses cipher replicating as a scheme for enhancing reverse engineering efforts.

Obfuscated software has become more complicated as a result of the employment of code clones. Baxter et al. [15] suggest AST matching as a method for detecting code clones. The necessity for reverse engineering might be eliminated by employing these methods to identify and eliminate code replicas. The Swap1 and Swap2 (collected by rearranging constants) duplicates in Figure 5 may be recognized by traditional methods, but the "The Memory swap" & and "XOR swap" duplicates in the code that comes next neither be.

```
/* Memory swap */
int
Swap_1(int x, int y)
{
    int t;
    t = x;
    x = y;
    y = t;
}
```

```
/* XOR swap */
int
Swap_2(int x, int y)
{
    int u;
    u = x ^ y;
    x = u ^ x;
    y = u ^ y;
}
```

Fig 5: Memory swap & XOR distinct clones

A developer must manually build the code clones at this stage. After the clones are built, in the subsequent spell you encounter a critical code snippet, check to see whether there is a structurally identical clone in the repository. If such a clone is discovered, the repository's nontrivial clones could be utilized to obfuscate the code snippet.

Step 3: Associating Code Clone Fragments:

Protection against dynamic analysis assaults is added in this stage. After constructing the nontrivial code clones, the original code fragments are replaced by the clones that are arbitrarily picked throughout every implementation of the associated novel code portion. Establish variable quantity is 'Boolean-valued variable star offered information (e.g. Collberg et al. [19]) assist solve the challenge. As Low [25] describes, such predicate variables introduce bogus control routes into programs. Static predicates are vulnerable to 'dynamic scrutiny outbreaks' since false rheostat pathways are on no occasion chosen throughout the execution of s/ware. Palsberg et. al [24] defined dynamic predicate variables as a resolute of an interrelated Boolean variable quantity. Variables have identical rates in one run of the software and have diverse standards on other scores of the s/ware.

In this paper, the obfuscation approach leverages a variation of 'dynamic predicate variables. These variables allow a specific mix of code clone pieces to be selected for a given run of obfuscated programs. They maintain dynamic structures (such as linked lists) to establish legitimate control flow pathways, as seen in Fig 5. This raises dynamic & and static analysis exertions since an assailant grasps the apprehensible outline.

To describe an HMM, the notations are shown below:

T = length of the observation sequence
N = number of states in the model
M = number of observation symbols
Q = {q0, q1, ... , qN−1} = distinct states of the Markov process
V = {0, 1, ... , M − 1} = set of possible observations
A = state transition probabilities
B = observation probability matrix
π = initial state distribution
O = (O0, O1, ... , OT−1) = observation sequence

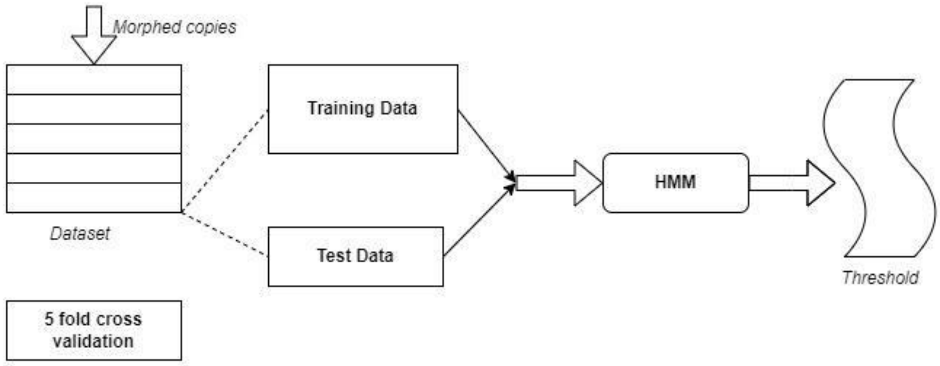

Fig 5: HMM Scheme

We do not demand that the final morphed code maintain the functionality of the original code to streamline the morphing procedure. To stop dead code from running, we inject it rather than including the necessary jump instructions. It deserves to be emphasized that this hinders recognition as a whole when an HMM-based sensor might effortlessly discern the altered source code distinct from the initial version due to the omitted jumping sequences. Furthermore, an upsurge in leap directives might represent an advantageous metric for detecting and eliminating code that has expired. The opcode sequence is compared to the previously determined threshold after being scored against the HMM model established during the training phase. If the allegedly fraudulent software's score surpasses the deadline, it implies it is sufficiently analogous to the actual program that it deserves further investigation. On the contrary side, an evaluation beneath the minimum threshold suggests that this skeptical software is unable to be recognized from authentic software. The detection procedure is depicted at a high level in Figure 7.

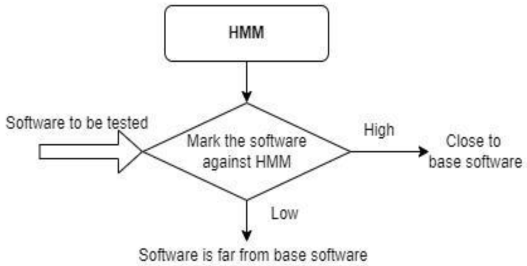

Fig 7: Detection Phase of HMM

Since the matrices A, B, and define a hidden Markov model, signify an “HMM as = (A, B,)”. The modules are depicted in Figure 5, with the "hidden" element situated above the dashed line. The existence of efficient approaches for dealing with each of the following challenges adds greatly to the strength and usability of HMMs. The proposed approach has two stages: aiming and recognition. During the training phase, a concealed Markov model is trained using subtly changed copies of the fundamental software. Comparison of the suspicious software to the model generated during the training phase during the detection phase.

IV.IMPLEMENTATION & RESULTS

The level of complexity of the disguised code doubles with the assortment of software replicas used for concealment, but a greater amount of manpower is required to generate the replicas. Developers oversee controlling the generation of complicated software replicas. If copies were reused throughout initiatives, the expense would be minimized since they would be amortized throughout the various initiatives.

Furthermore, the proposed approach will result in minor performance loss because:

- Using dynamic predicate variables, code clones are connected. The proposed design of dynamic predicated variables was demonstrated to be computationally cheap. Static analysis assaults therefore are defeated. In obfuscated code, the proposed obfuscation approach familiarizes an ‘exponential no.’ of legal controller stream pathways.
- If the obfuscated code has 'N logical code fragments' and 'K clones per fragment', it will comprise 'KN routes', every one of which correlates to a specific distinct route in the source code. As a result, an assailant must implement software several whiles to gather traces and hence withstand dynamic analysis techniques. Finally, the clones' structural dissimilarity opposes static clone detection approaches.

Table 2: Comparison of Original & Obfuscated Code

Metric	[Ori, Tob, NTOb] sortData()	[Ori, Tob, NTOb] searchData( )
{LOC}	[23,432,67]	[9,181,50]
{10KB} Data	[7s,7s,8s]	[1s,1s,1s]
{25KB} Data	[110s,110s,115s]	[2s,2s,2s]
{50KB} Data	[540s,542s,553s]	[7s,7s,7s]

searchData() & sortData() are the search & sort functions of Figure 6's data processing program. Whereas, the 'LOC' depicts ‘Lines of Code’, while the following three rows represent the execution times with input varying from 10KB to 50KB in size for the Data Processing Application. NTOb, Ori & and TOb represent the novel package, the program obfuscated using inconsequential code replicas, & and the software package obfuscated with program replicas, respectively. Conclusions demonstrate that whereas there is no substantial degradation in performance for the information handling implementation, our approach traverses the Cost demand. The computation times on a "64-bit Windows 10 computer with 4 GB of RAM and a 2.1 GHz dual-core processor" are shown in Table 2. We examined three other parameters in addition to performance, which are depicted in Table II. This table demonstrates the cyclomatic intricacy of the novel software and the programming obfuscated using nontrivial code clones. Cyclomatic complexity quantifies the amount of self-regulating directions. The PMD tool was incorporated to determine this. The cc of an obfuscated program is ‘>’ the cc of a novel program. This suggests that the obfuscated s/ware is resilient, and hence our technique meets the Resilience requirement. Also, the increased Cyclomatic complexity & and LOC provide substantial obstacles for even an individual developer of the obfuscated code. Hence, the proposed approach encounters the Effectiveness criteria.

Table 3: Measurement of Other Parameters

Metric	sortData() [Ori, NTOb]	searchData() [Ori, NTOb]
Cyclomatic	[5,19]	[3,16]
Coverage	[100%, 100%]	[100%.100%]
Memory	[4MB,4MB]	[4MB,4MB]

The row Coverage in Table 3 represents the basic block coverage. The EMMA tool was incorporated to determine coverage. The findings reveal that whole essential code-blocks of were executed, indicating replicas were implemented throughout a specific iteration of the other's code. This shows that the obscured s/ware comprises all valid control flow routes. The proposed obfuscation approach is impervious to “dynamic analysis efforts” since the no. of allowable regulator flow routes in the context of replicas of code & logical segments of code is limitless. How much RAM the original & disguised programs used were also measured. The proposed approach has no major memory overhead, as evidenced by the 'Memory' row in Table 3. Lastly, the tools on code obfuscated were tested with simple clones & code wrapped with complex replicas to verify if replicas could be encountered programmatically. Every single one of the aforementioned techniques recognised basic code clones. This illustrates how challenging it is for an intruder to recognise non-trivial software copies.

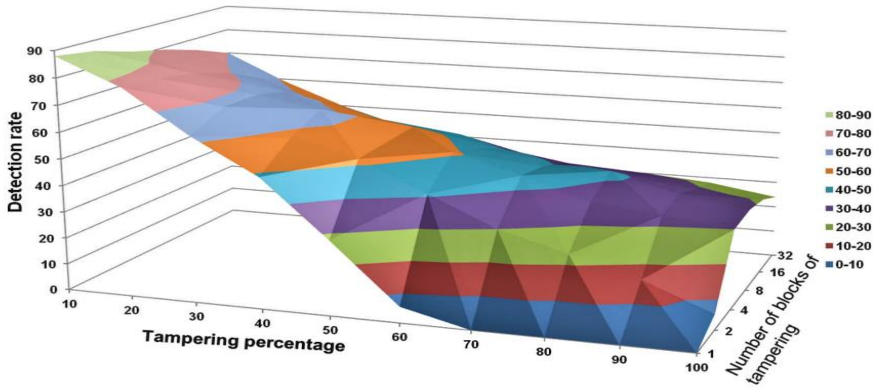

Fig 8: Results

- The opcode classification from the undone base file was collected, and 100 modified replicas ignoble folders were produced, by 10% altering and 1 block of deceased cipher. To avoid the HMM from overfitting the training data, modified versions of the basic program are used.
- Hidden Markov models with five-fold cross-validation were trained using the 100 morphing copies.
- Relying on the results for a sample of 15 "normal" documents that were not used in the warping or instruction plus a set of morphing files that were not employed in the training, a criterion was established. The threshold had been established at the highest achievable score for any of the "standard" files.

As predicted, detection success falls as the rate of tampering increases. Unexpectedly the quantity of tampered wedges has a substantial influence identification stage, specifically when tampering frequencies are high. Figure 8 demonstrates that "1-block tampering significantly influences rising rates, with 70% or more tampering; none of the 1-block tampered files are accurately identified".

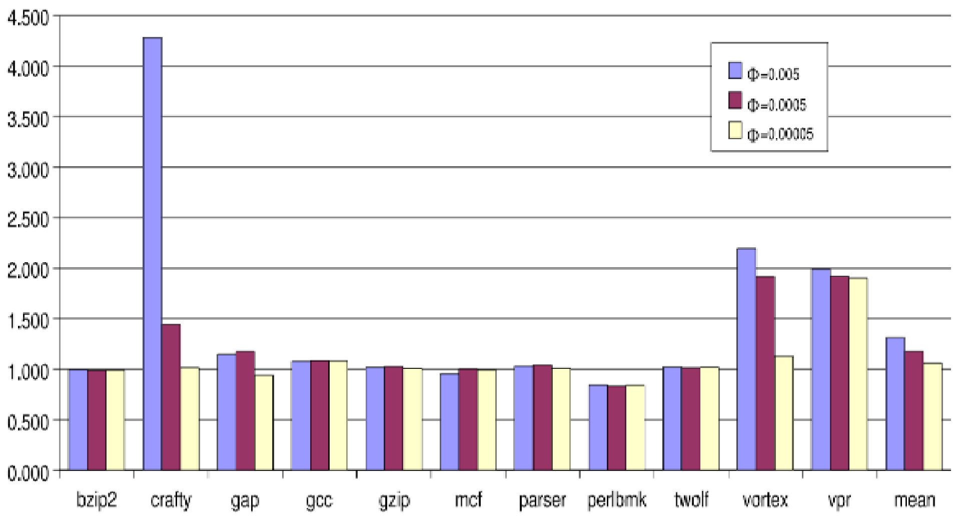

Fig. 9 depicts the slowdown in execution time associated with varying values of φ.

Figure 9 provides a comprehensive overview of the run-time overhead incurred by our transformations across various values of φ. The average performance degradation for benchmarks is observed to be 33.1% when φ is set at 0.005, and a comparatively lower slowdown of 6.9% is noted when φ is adjusted to 0.00005.

V.CONCLUSION

In this research, an obfuscation approach was introduced for protecting critical software code pieces. Code obfuscated by the proposed technique meets the efficacy standards specified in the manuscript, subject to the eminence of the s/ware replica outlines. Even though the approach necessitates surplus building costs for the replicas, it appears to be beneficial for obscuring subtle software portions such as information concealing & and licensing verification.

Furthermore, although this paper presented the strategy for obfuscating C++ and Java programs, the outline is relevant to s/ware inscribed in any imperative language, counting C & and C++. We intend to test our technique on huge industry codes.
As of now, a functioning sample is complete, it needs supplementary execution assistance. As soon as established, will allow to conduct experiments to better comprehend the real-world challenges intricate in implementing the proposed approach. The suggested approach of S/ware Fortification is safe contrary to recognized intimidations of assault (Man-in-the-Middle attack, Brute force attack, and replay attack) by employing upgraded MD5 & RSA encryption techniques & and a combination of cryptoanalysis procedures. The experimental findings reveal that our technique is resilient in the sagacity that the underlying program might be heavily updated formerly failed to categorize it with a high prospect. In testing, the conclusion is that morphing is extremely effective at lowering the HMM scores, that the attacker selects the morphing code in the best possible way, i.e., from files that are the same as those used to calculate the threshold. Furthermore, we neglect to take into consideration how challenging it would be for the assailant to preserve the code operating whilst morphing. The "morphing" code might have a few alterations constraining the evolving possibilities, knowledge about the files that are utilised for the thresholding would prove diligently for someone to come by, & and preserving the intended features of the source code would ultimately be difficult at times. In practice, the perpetrator would ultimately be at a substantial disadvantage in these areas of the code.
The findings of this study suggest that inserting a huge chunk of deceased programs is the best method for an invader. However, the efficacy of such a method may certainly be countered by utilizing some of the approaches outlined. Further morphing approaches and more tests on a wider range of file formats might be used in future development. Furthermore, because '1-block tampering' might be a successful assault technique, additional tests targeted to reduce the efficacy of such an attack would be beneficial.

REFERENCES:

1. BSA website http://www.bsa.org/country/Anti-Piracy/What-is-Software-Piracy/Types%20of%20Piracy.asp x, 2007.
2. T.Premkumar Devanbu and Stuart Stubblebine, "Software engineering for Software Engineering for security a roadmap" ,In Proceeding of the conference on the future of software Engineering,pages 227-239.ACM Press,2000.
3. Bruce Schneier. " Schneier on Security: Sony's DRM Rootkit: The Real Story", http://www.schneier.com/blog/archiv es/2005/11/sonys_drm_rootk.html, 2005.
4. "Concepts of Cryptography," URL:http://www.Kremlinencrypt.co m/concepts.htm.
5. "Encryption Algorithms",URLhttp://www.networksorcery.com/enp /data/encryption.htm.
6. Wikipedia, "The Free Encyclopedia, Data Encryption Standard," URL: http://www.wikipedia . 2(28), 44-60, 1997.
7. Smith,Don Copper,"The Data Encryption Standard (DES) and its Strength Against Attacks", IBM Journal of Research and Development (1994), 38(3), p243-250.
8. Zoeller & Renate. " Nest of Pirates", Transitions Online, p.5-5, 2007.
9. Mustafa Al-Fayoumi, Sattar J. "An Efficient RSA Public Key Encryption Scheme", Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference, p.127- 130, 2008.
10. Collberg, C.S.; Thomborson, C.; "Watermarking, tamper-proofing, and obfuscation - tools for software protection", IEEE Transactions on Software Engineering, Volume: 28 , Issue: 8, Page(s): 735 – 746, 2002.
11. J. Cappaert et al., "Toward Tamper Resistant Code Encryption: Practice and Experience," LNCS, vol. 4991, 2008, pp. 86-100.
12. H. Chang and M. J. Atallah. Protecting software codes by guards. ACM Workshop on Digital Rights Managment (DRM 2001), LNCS 2320:160-175, 2001
13. Hongxia Jin; Lotspiech, J.; "Forensic analysis for A Survey on Software Protection Techniques against Various Attacks © 2012 Global Journals Inc. (US) Global Journal of Computer Science and Technology Volume XII Issue I Version I 57 January 2012 [14] Presented a code block encryption approach to protect software using a key chain tamper resistant software", 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.
14. D.W Jung, H.S Kim, and J.G. Park, "A Code Block Cipher Method to Protect Application Programs From Reverse Engineering,"J. Korea Inst. Inf. Security Cryptology, vol. 18, no. 2, 2008, pp. 85-96 (in Korean)
15. Song-kyoo Kim; "Design of enhanced software protection architecture by using theory of inventive problem solving", IEEE International Conference on Industrial Engineering and Engineering Management, 2009. IEEM 2009.
16. B. Horne, L. Matheson, C. Sheehan, R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance", Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp.141–159 (2002).
17. P. Gutmann, "An Open-source Cryptographic Coprocessor", Proc. 2000 USENIX Security Symposium.
18. E. Eilam, Reversing: Secrets of Reverse Engineering, Wiley Publishing, Inc., 2005

19. C. Collberg, C. Thomborson, and D. Low, ”A taxonomy of obfuscating transformations”, Technical report 148, Department of computer science, the University of Auckland, New Zealand, 1997.
20. T. J. McCabe. ”A complexity measure”, IEEE Transactions on Software Engineering, 2(4):308-320, December 1976.
21. M. Madou, B. Anckaert, B. D. Sutter, and D. B. Koen. ”Hybrid static-dynamic attacks against software protection mechanisms”, In Proceedings of the 5th ACM Workshop on Digital Rights Management. ACM, 2005.
22. I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In Proceedings of ICSM. IEEE, 1998.
23. S. Schrittwieser and S. Katzenbeisser, ”Code Obfuscation against Static and Dynamic Reverse Engineering”, Vienna University of Technology, Austria, Darmstadt University of Technology, Germany
24. Palsberg, J., Krishnaswamy, S., Kwon, M., Ma, D., Shao, Q., and Zhang, Y.: ”Experience with software watermarking”, In Proceedings of 16th IEEE Annual Computer Security Applications Conference. IEEE Press. p308. New Orleans, LA, USA. 2000.
25. Low, D. (1998). ”Java Control Flow Obfuscation”, Master’s thesis. University of Auckland.
26. S. K. Debray and W. Evans. Profile-guided code compression. In Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02), pages 95–105, June 2002.