SN Computer Science (2023) 4:10
https://doi.org/10.1007/s42979-022-01433-1

ORIGINAL RESEARCH

A Novel Framework for Metamorphic Malware Detection

Animesh Kumar Jha[1] · Abhishek Vaish[1] · Sairaj Patil[1]

Received: 28 November 2021 / Accepted: 25 September 2022 / Published online: 15 October 2022

Abstract
Malwares are a major threat in the evolving global cyberspace. The different techniques for anti-virus software, in which presently there is insufficiency in detecting metamorphic malwares as they can change their internal structure of the code, keeping the flow of the program equivalent to the virus. Commercial Antivirus software depends on signature detection algorithms to identify viruses, but code obfuscation techniques can circumvent the above algorithms successfully. The objective of this research is to analyze the different detection techniques of such metamorphic malware. We also propose a novel methodology of detecting them via use of different machine learning algorithms, such as KNN, Support Vector Machine (SVM), RF (random forest), and naive Bayes. We also establish multiple semantic preserving transformation techniques for code obfuscation. Analysis regarding the same has been presented too.

Keywords Metamorphic malwares · Semantic preservation transformation · Code obfuscation

Introduction

Malware is a software program deliberately designed to cause harm to computer networks, servers, clients or computer systems with no user consent. The main motive of these malware is to get important information from one's system and also to use resources with no user consent. These malware can slow down users' computer systems, crash computers, steal private information, send spam through and to inbox, infect computers, and use it to transfer various files or attacks.

Malware is an evolving and ever-growing threat for the global cyber space. The number of malwares detected each year is constantly increasing and so is their ability to circumvent detection techniques. New forms of morphic malwares are emerging that are capable of changing their signatures and evading detection by signature-based algorithms.

The figure highlights the constant growth in malware production in the past decade. While in 2020, the number of malwares reported was 1273 million [1], the numbers of "never-seen-before" malwares were only 268,000 [2], and total number of new malwares was also around 146 million [1]. The research by Camponi et al. [3] also reveals that over 66% of the malwares are morphed from previously known threats. This highlights the fact that the maximum number of malwares in circulation is either old malwares or morphed versions of the same. While we have techniques to easily detect the older variants using signature-based mechanisms and some newer ones using emulators or DFA-based techniques, the morphed variants are the most troublesome of all due to their slippery nature.

Further, the research report by Ponemon institute on 2018 state of endpoint security risk, has also concluded that at least 76% of the organizations are totally dependent on commercial anti-virus programs that use signature detection technique to detect any intrusion or vulnerability. This in turn, makes the organizations weak against metamorphic malwares, and makes a compelling argument for research and development in the detection of same.

These malware have a mutation engine that takes in the code as input and return a morphed version of the code in each iteration. The morphing is done on the basis of semantic preservation techniques, which ensures that while the code signature changes, the effect of the code essentially remains the same. It can alternatively be said that, in metamorphic viruses, the physical appearance of the source code is morphed, keeping the logical flow the same. This in turn changes the hash of the signature code of the code. Hence,

✉ Animesh Kumar Jha
animesh0906@gmail.com

[1] Indian Institute of Information Technology Allahabad, Prayagraj 211012, India

it is difficult to detect these metamorphic viruses using signature detection algorithms.

Different techniques have been found for the detection of malwares. Most commonly used technique is signature-based algorithms and used by most of the commercially available anti-virus software. Signature-based algorithms use the physical structure of malwares to distinguish malwares. They use a database of malware signatures to detect potential malwares. Metamorphic malwares can change their physical structure keeping the same control flow, thereby making it difficult to detect with the signature-based techniques. Some other techniques for detection are:

Behavioral Detection

It uses the dynamic nature of the malware rather than the conventional static way of signature detection. Extraction of dynamic behavior is done by executing a malware file in isolated surroundings [4]. The main advantage of the behavioral detection is when a computer virus looks similar to a benign program, but its functional aspect can describe it as harmful program. Here the above technique along with the machine learning algorithms can be used to classify a record into harmful malware or a normal program. A research by P. Desai, concludes that a metamorphic malware code can be similar to a benign application by nearly 93%, making it way more close to benign files than malwares [5].

Anomaly Detection Technique

Anomaly detection chips away at the methodology of identifying if the document present follows a typical behavioral aspect. It beats the restrictions created by signature-based detection algorithms utilizing heuristic-based ways to deal with recognizing ordinary behavior. If a file is not classified as a normal file, then it is classified as a harmful malware. Here the notion of anomalous and normal behavior is expounded by the user. Hence, it does not give accuracy in classification. Malware detection emulator was created to distinguish the input programs into different classes based on the appearance of the record. Once the classification is done, then it is reviewed whether it is harmful malware or an exceptional false positive case of being a malware.

In this research, we try to morph malware codes and later create a classifier using machine learning algorithms to detect these morphed malware files. We also go on to discuss various morphing techniques that can be employed to bypass the detection and analyze accuracy drop for the same. The subsequent sections deal with existing work and literature review, followed by a proposed novel methodology for detecting metamorphic malwares.

Literature Review

Campion et al. [1] have proposed an analysis on the schema and working of Metamorphic malwares by attempting to develop a framework that can produce the original variant of the malware by analyzing different samples of the same malware and detecting the semantic preserving transformation rules applied to obtain original malware code. This exercise also helps bring in insights about the Metamorphic malware engine, or the Mutation engine. The research reveals that on an average, over 90% of the code in the metamorphic malwares is that of the engine itself. The engine serves the entire code as its input, and a morphed version is thrown as output, using various transformations that essentially keep the semantics same. The engine also has a definite structure with various parts:

(1) Dissembler–converts code to assembly instructions.
(2) Code Transformer–Applies code obfuscation techniques.
(3) Assembler–converts mutated code to binary instructions.

The research by Kakisim et al. [4] revolves around generating a framework for detection of metamorphic malwares using automata principles. It establishes that every version of metamorphic malware is different from all others, and the possible number of versions is so huge that a database cannot be maintained, thereby removing the possibility of using a commercial anti-virus software or using any pattern matching algorithm. The authors have suggested using the control flow of the program rather than the code itself. They attempt to develop an opcode (instruction machine code) graph and then superimpose different versions of it to detect similarities between different malware to find the engine code. They use the assumption that different variants formed out of the same malware will have a common engine-specific pattern. The problem in this methodology that was observed was that they needed to have multiple samples of the same malware variants and then would be able to predict the malwares that had the same engine-specific variants. The possibility of detecting a totally new variant is not prevailing in this research.

William B. Andreopoulos [3] has used the assumption that a program is basically a set of instructions that are executed in a sequence and has therefore used sequence-based Machine learning to derive insights to detect malwares. The proposed framework has been established to work in case of morphic malwares too, i.e., polymorphic and metamorphic malwares. They have also used Hidden Markov Models (HMM) principles and Long Short-Term

Memory (LSTM) networks. The proposed model is a dynamic approach that can be incorporated in the antivirus systems to provide real-time detection of viruses. Sequential features extracted from malware source code analysis have been used for the classification of malware with deep learning approaches.

A novel approach has been proposed by Javaheri et al. [6] to use genetic algorithms for detecting future variations of targeted and metamorphic malwares. The researchers have extracted a sequence of system API calls using various filters. The authors created a sandbox environment and unpack and execute the files and go for a sophisticated behavioral analysis to ascertain the classification of the file as harmful or benign. Dynamic unpacking was performed which is based on kernel-level memory dumping. Once unpacked, the malware features were extracted by parsing it, to find relevant sequence flows of instructions to model the malware behavior. The malware is further executed in a virtual box environment and its activity, tracked and recorded. Finally, Genetic Algorithm (GA) was employed, taking in each behavior as a chromosome and each system call mapped to a gene. They used linear regression to model both the behavior and chromosome formation. The research suffered a major drawback in the form that it needed 2 + samples for generating variants and performing crossovers. Also, recent metamorphic malwares have been shown to hide original behaviors in a contained environment and therefore feature extraction can fail.

Data mining techniques have been used for detection and classification problems from a long time, and Souri et al. [7], in this research work, have tried to summarize different data mining techniques used to predict malwares and presented an analysis on the same. The paper only looks to classify approaches as signature-based or behavior-based, and has not dealt with evolving technologies, such as Genetic algorithm, Sequence-based models, Hidden Markov Models and Graph-based approaches, making the scope of the analysis limited.

Hidden Markov Models' use for malware detection has evolved as a popular research domain. It is because of the fact that any computer program can be represented as a sequence of instructions, thereby treating a program as a time series, an ideal condition for usage of Hidden Markov Models [8].

Annachhatre et al. [2] have applied HMMs and Cluster Analysis to detect unknown variants of malware by analyzing the control flow of the program. The HMM is trained on the basis of given observation sequences. Initially, HMM is trained for a variety of compilers and Malware generators, training involves both, forward and backward algorithms and scoring is done by the forward algorithm. Finally, the clustering is done using K-means. The implementation is a straight-forward HMM scoring system, and it is proven that for unknown samples on which HMM was not trained, it was still able to identify and classify them with decent accuracy. Modifications on this can be made to increase the accuracy and usage of Fuzzy neural networks as a future scope.

Toderici et al. [9] have shown that Metamorphic malwares can evade detection by HMM if they use morphing with instructions from benign files. They have alternatively proposed a chi-squared-based solution coupled with HMM to detect metamorphic malwares. The chi-squared distance is calculated based on the differences in instruction opcode frequency, which has shown a higher accuracy compared to [2].

Devendra et al. [10] have presented a detection technique for metamorphic malwares using machine learning techniques. They have proposed the use of a Support Vector Machine (SVM) as a tool to detect such malware variants, using a specialized kernel for the model, called the Histogram intersection kernel. The kernel allows us to fetch an optimal hyperplane for differentiating between the malware variants and harmless files. The histogram generation is done on the basis of opcode frequencies and then is normalized to minimize effects of obfuscation techniques. The classifier tries to detect the base malware of which the variant can be a part of. The main issue behind the problem is that in case if the morphed variant is itself a variant from a never-seen-before class of malware, it will not be possible to detect it as the base malware is not known to us.

Kancherla et al. [11] have also proposed use of SVM for malware classification. They have used the malware executable to be transformed into an image called the bytecode image, and then intensity-based and texture-based features are extracted to predict the code signature. SVM is then employed for bifurcation of the dataset into benign and malware signatures. An accuracy of 95% is reported in the paper. Code obfuscation and morphing have not been incorporated, therefore, limiting its applicability for metamorphic malware detection.

In the proposed implementation of this research, we have tried to work on a very diversified and extensive dataset. Unlike the existing modules that work on a specific malware type and then detecting morphed variations of it, the dataset used is composed of different malware and benign files which ensures a higher degree of diversification, which in turn leads to a better confidence for detecting zero-day malwares. Also, the model is replicable for multiple post processing algorithms for training and testing and analyzing the detection accuracy.

Comparative Analysis Based on Literature Survey

In this section, we will be presenting an analysis table of different detection techniques for metamorphic malwares.

Proposed Architecture:

Our proposed system consists of three layer:

(1) The Morphing Engine Layer.
(2) Transformative Layer.
(3) Classification Layer.

We shall discuss all three layers separately in the following sections.

Morphing Engine Layer

The objective of this part of the system is to create a dataset of the Morphed code which is fed to the detection classifier, it can be seen in Fig. 2. We have used 150 k benign files of Jscript records as a benign dataset [5]. Preprocessing is done to reduce redundancy, after which feature extraction is performed. We have used these 40,000 malware records and the 40,000 records for the process of training the machine learning models for the detection of metamorphic malwares (Table 1).

For the process of morphing of the dataset files, we use techniques like inserting a dead code, renaming the variable declarations, changing the position of the functions, and the reordering of the instruction. This will get us 40,000 files of morphed codes i.e., records of morphed files.

Morphing Implementation Details

Abstract Syntax Tree Data Structure

Here input parameters are javascript code and output result is Abstract Syntax Tree(AST). We transform code into AST using Esprima nodeJS module [3].

Code Obfuscation

The AST obtained from the previous step is morphed with the morphing techniques discussed later. We get the AST from Esprima in the json format, and for the morphing process we, change the key-value pair of json.

Convert AST to Code

The AST obtained from the previous step is again transformed into the JS code using the escodegen module. The main essence of escodegen is to reverse the function aspect as it transforms the AST to code. For e.g.,

Following is the code and below and its Abstract syntax Tree is given.

Figure 3 shows (AST) Abstract Syntax Tree of the above JS code. In the process above, the tree is then transformed into the machine language code. Our main intention here is to morph the code using Abstract Syntax Tree as we have a logical structure of the code. We can change the physical appearance of the code keeping the same logical structure.

The final morphed dataset is tested to ensure the efficacy of the Morphing Layer. Therefore, the morphed dataset is tested using different ML techniques and results are compared. A comparative study is done for 2 different datasets, one with base malware files, and the other with morphed files to check the accuracy of the drop.

Techniques used for Code Obfuscation

In this section, we will see some techniques for code obfuscation that allows semantic preservation while altering the signature.

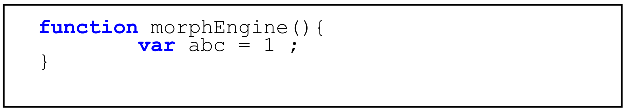
Renaming of Variable Declarations

Here, we can change the names assigned to the variable declarations keeping the compilation parameters the same. In Jscript notion, there can be function within function, i.e., inner function and they both can have same named [3] variables. Hence, we need to be careful to change variable [2] names to solve this problem, we use stack as data structure [12].

Insertion of Dead Code

Here, we try to insert the dead code into the original code. Dead code here refers to codes that are not going to contribute to the main working of the program, but are added to increase the size of code and can often be parts of codes from benign files, which in turn transforms the signature of the source code without transforming the logical flow of the program.

Code 4.1 (Original Code)



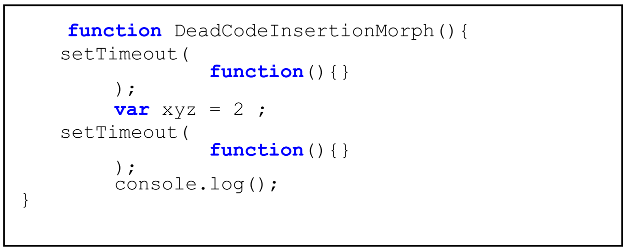
```
function morphEngine(){
        var abc = 1 ;
}
```

Here is an example, in our case, we insert no meaning instructions like no return function no value function, etc. explained in code 4.1, we can also add set timeout or console log instructions. Above method does change the signature of the original code.

Table 1 Analysis of detection techniques for metamorphic malwares

Detection technique	Methodology	Advantage	Shortcomings
Signature-based detection	Matches the signature of present file from a database of expected malware signatures using (1) Using regular expressions (2) Pattern matching	(1) Simple, Fast and Efficient (2) Effective against normal virus (3) Higher accessibility	(1) Needs regular updates in the database (2) Zero possibility of detection in case of a new virus not present in database (3) Generally cannot work against metamorphic virus due to constantly changing signature (4) Database management and storage overhead
Heuristic-based method	(1) Examines code for possibility of suspicious code (2) Can be used in static, dynamic or hybrid method	(1) Can detect never-seen-before malwares (2) Capable of detecting polymorphic and metamorphic virus	(1) High rate of false positives (2) Can be circumvented by metamorphic malwares if executed in controlled environment (3) High time complexity for detection
Hidden Markov model	(1) Uses probability of transition from one state to other (2) It is trained over different compilers and generators to obtain state transition information (3) Scoring system is developed to score and finally classify the codes as different types of mal- wares	(1) The model is competent to detect never-seen-before malwares (2) Scoring methodology can be varied for higher accuracy attainment	(1) Complexity is high (2) Accuracy is not high enough for commercial usage (3) Fails to detect metamorphic mal- wares if a training benign file is used for dead code insertion
Support vector machines	(1) SVM can be applied once the dataset is morphed into a meaningful form (2) Executable can be converted into bytecode images or opcode frequency histograms	(1) Simple to implement once pre-processing is done (2) High accuracy is achieved when used to classify (3) Different kernels can be used to increase accuracy	(1) Pre-processing is lengthy (2) It is a supervised algorithm, newer variants may go undetected
Emulators	(1) Code emulation implements a virtual machine to simulate the CPU and memory management system (2) It executes malicious code inside the virtual machine	(1) Performs dynamic analysis and studies effects of malware using sand- boxing (2) High accuracy can be seen even for morphic malwares	(1) Time complexity is very high (2) Newer mutation engines are capable of suppressing dangerous behavior if closed environment is detected

Code 4.2

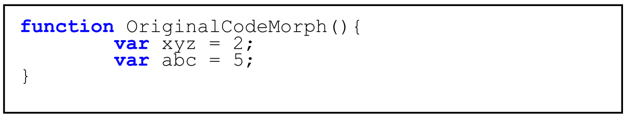


```
    function DeadCodeInsertionMorph(){
   setTimeout(
                 function(){}
       );
       var xyz = 2 ;
   setTimeout(
                 function(){}
       );
       console.log();
}
```

Changing the Order of the Instructions

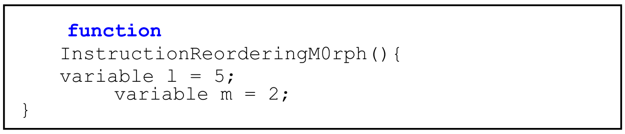
Here, we try to search the definition of the variables which is declared using Abstract Syntax Tree (AST) and we try to ensure these instructions are independent and have no effect on the logical flow of the original code.

Code 4.3



```
function OriginalCodeMorph(){
        var xyz = 2;
        var abc = 5;
}
```

Code 4.4



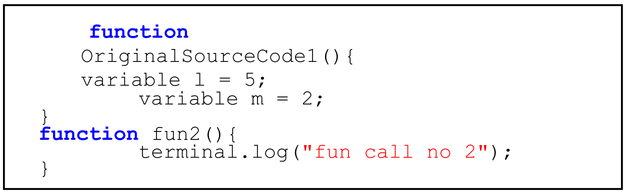
```
    function
   InstructionReorderingM0rph(){
   variable l = 5;
        variable m = 2;
}
```

One of the examples of instruction reordering is we try to identify the variable declaration from (AST), then we interchange the statement described in code 4.3; once the above process, we get the changed code as in code 4.4 above process does not transform the logical flow of the original source code.

Changing the Position of the Function Declaration

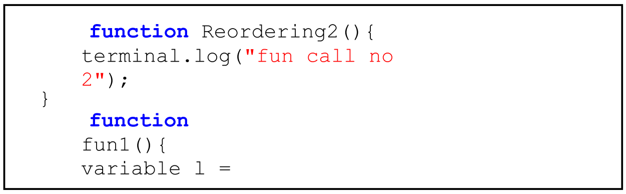
In this method, we try to reorder the function as we did in instruction reordering. Let us suppose we have k no. of different functions in a code we can permute them in $k!$ as explained in code 4.5.

Code 4.5



```
     function
    OriginalSourceCode1(){
    variable l = 5;
         variable m = 2;
 }
 function fun2(){
         terminal.log("fun call no 2");
 }
```

Code 4.6



```
     function Reordering2(){
    terminal.log("fun call no
    2");
 }
     function
    fun1(){
    variable l =
```

Arranged in $2! = 2$ ways.

Hence, output after the function reordering would be seen as in code 4.6

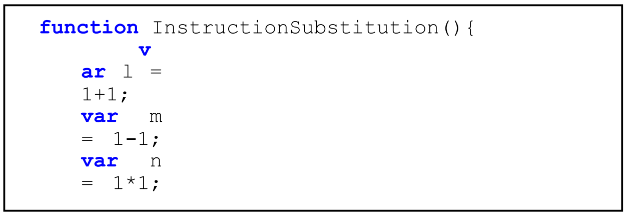
Substitution of Instructions

In the above method, we try to substitute the arithmetic operators with other operators.

Below is the example:

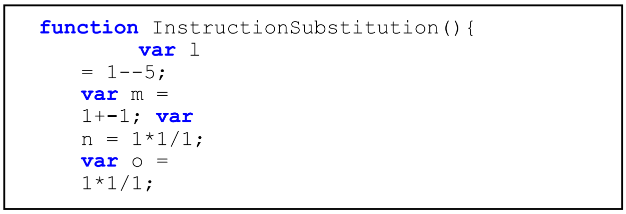
When we try to sum 2 numbers, for example, let us take an expression $vari = 1 + 1$, it can be replaced with $vari = 1 - -1$, which will give us the same output. The source code is in code 4.7 and this code transforms to code in code 4.8.

Code 4.7



```
 function InstructionSubstitution(){
          v
    ar l =
    1+1;
    var   m
    = 1-1;
    var   n
    = 1*1;
```

Code 4.8



```
 function InstructionSubstitution(){
          var l
    = 1--5;
    var m =
    1+-1; var
    n = 1*1/1;
    var o =
    1*1/1;
```

Transformative Layer

We have used Rhino for this layer. Rhino is a project created by Netscape which is a Java writer JScript engine. Rhino is commonly used to transform the JS code to classes based on java language and it can be executed in either compiler mode or interpreter mode. The rhino software helps us get the bytecode of these.js files. This in turn helps us get the opcode sequence files from the bytecode, opcode will specify the arithmetic to be done for the program. We get the sequence file for all the dataset. After this, we try to extract features for the training ML classifiers.

Classification Layer

The main role of this layer is to classify metamorphic malware from a lot of given records. Our aim is to classify such metamorphic malware records that are classified benign. In other words, to detect the code which is morphed from the first source code is a malware file.

Dataset

As already indicated, we are proposing a three layer architecture. The output of the first layer i.e., the Morphing Engine Layer is fed to the classification layer. This can be seen in Fig. 1, and the data specification can be referred from Morphing Engine Layer.

Extraction of the Features

In the above section, we provide a sequence file as input. Our aim is to extract features and train these features in machine learning algorithms. We use the n-grams technique to extract features from the sequence file (Figs. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14).

N-gram Extraction

In the above method, we keep a window of words of arbitrary length and we try to find the frequency of the coexisting words. We move the window to get the frequency of words which are occurring in the same window. This technique is used in the NLP and data mining context.

Below is the snippet of the N-gram from the sequence file where $N=5$.

Following are the steps involved in the N-gram feature extraction:

In our research, we have used the 100 length vector for feature extraction and training the machine learning classifiers.

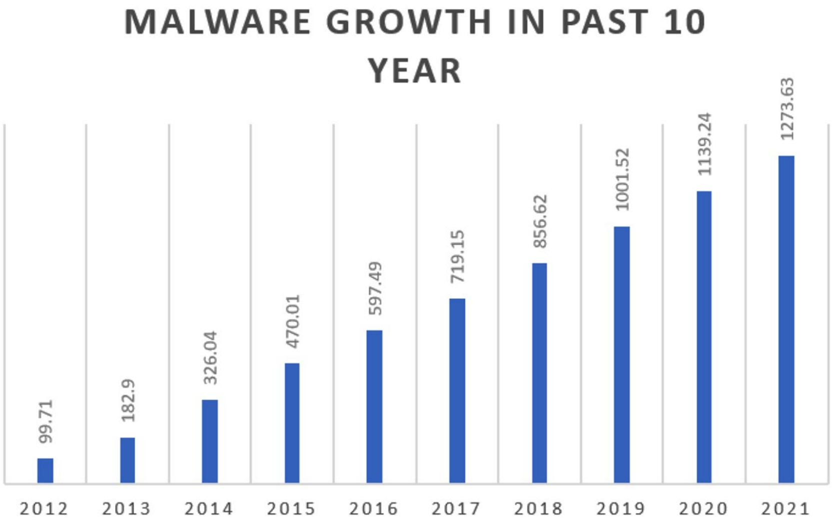
Training with N-gram

Once we are done with the process of the extraction of the features. We try to train these features using machine learning classifiers. We used 80–20 splits for cross validation. Here our input parameters are N-gram vectors for the purpose of training. We are using supervised learning techniques like Support Vector Machine, Random Forest, KNN and Naive Bayes.

Experiments

In the above section, we look into the experiments that are done for the research. The setup is made in a virtual box for feature extraction and pre-processing on the malware files. Later the computing is done on host system.

Fig. 1 Overall development of new malware programs over last 10 years



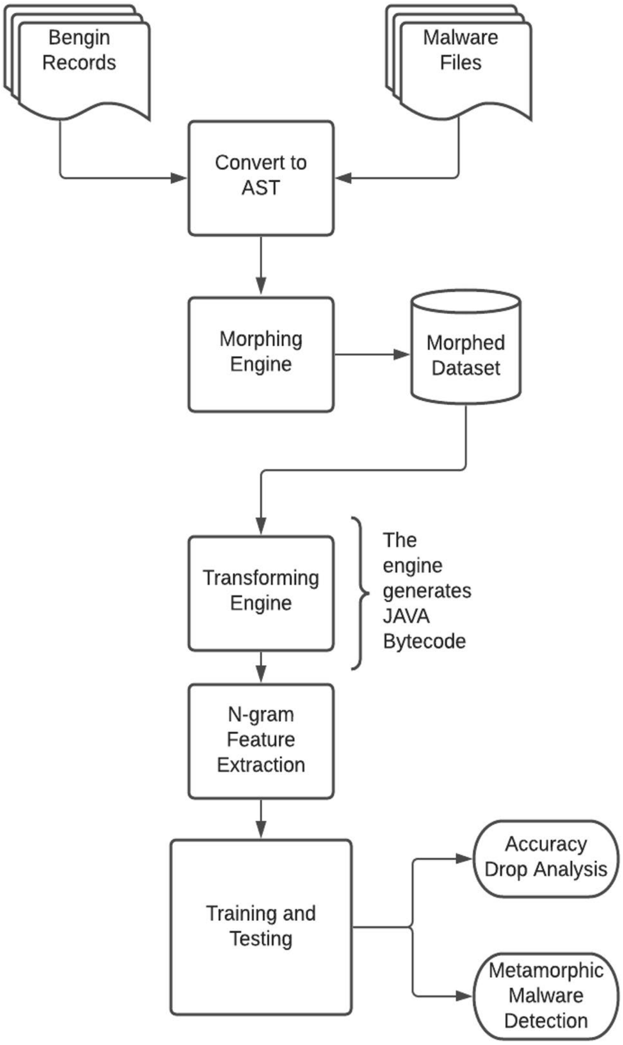


Fig. 2 Proposed Architecture

Results

Our first aim is to decrease the precision of the detection of malware systems by introducing morphing. Here we used the N-gram feature extraction technique. First aim can be fulfilled using morphing done to the source code as discussed in earlier sections.

Following are the results based on the algorithms used.

Random Forest

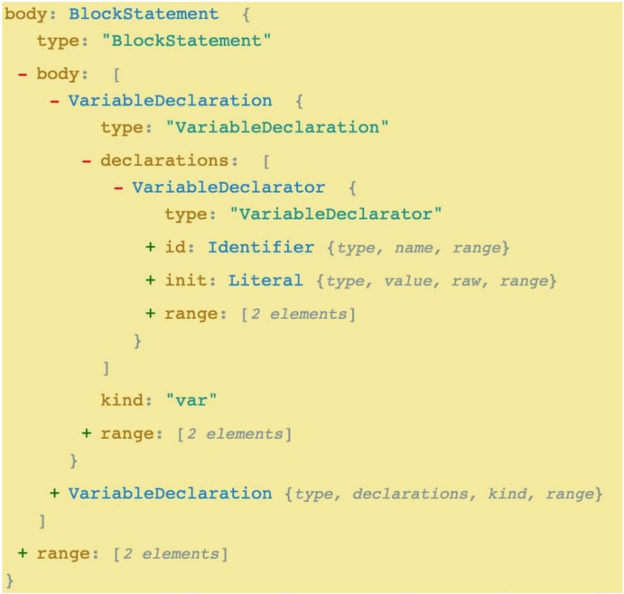
$AUC-ROC=93\%$

Trees Used For our algorithms = 100.

K Nearest Neighbors

$AUC-ROC=94\%$

Value of k is 5.



```
body: BlockStatement  {
   type: "BlockStatement"
 - body:  [
    - VariableDeclaration  {
         type: "VariableDeclaration"
       - declarations:  [
          - VariableDeclarator  {
               type: "VariableDeclarator"
             + id: Identifier {type, name, range}
             + init: Literal {type, value, raw, range}
             + range: [2 elements]
            }
         ]
         kind: "var"
       + range: [2 elements]
      }
    + VariableDeclaration {type, declarations, kind, range}
   ]
 + range: [2 elements]
}
```

Fig. 3 Abstract Syntax Tree of given code

Support Vector Machine

$AUC-ROC=96\%$

We use linear models for classification.

Naïve Bayes

$AUC-ROC=94\%$

In the above comparison, SVM technique shows the best results for detection of the malwares i.e., 96% after applying the morphing techniques, it reduced to 74%, following is the comparison after introducing the morphing techniques.

Results with Different Input Parameters

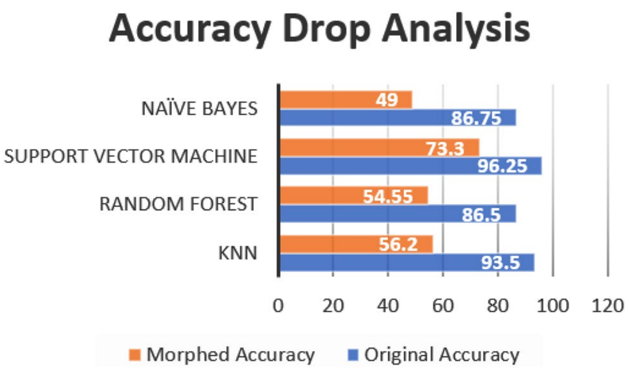
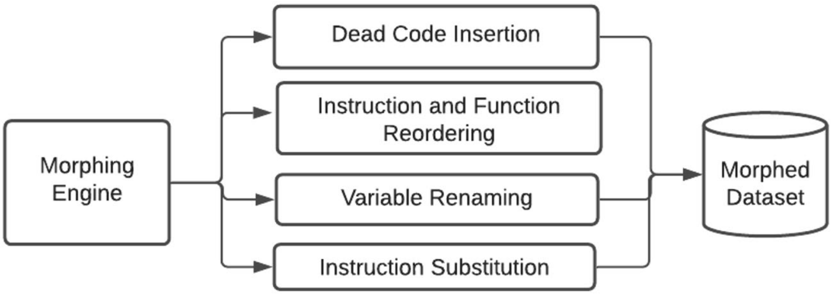
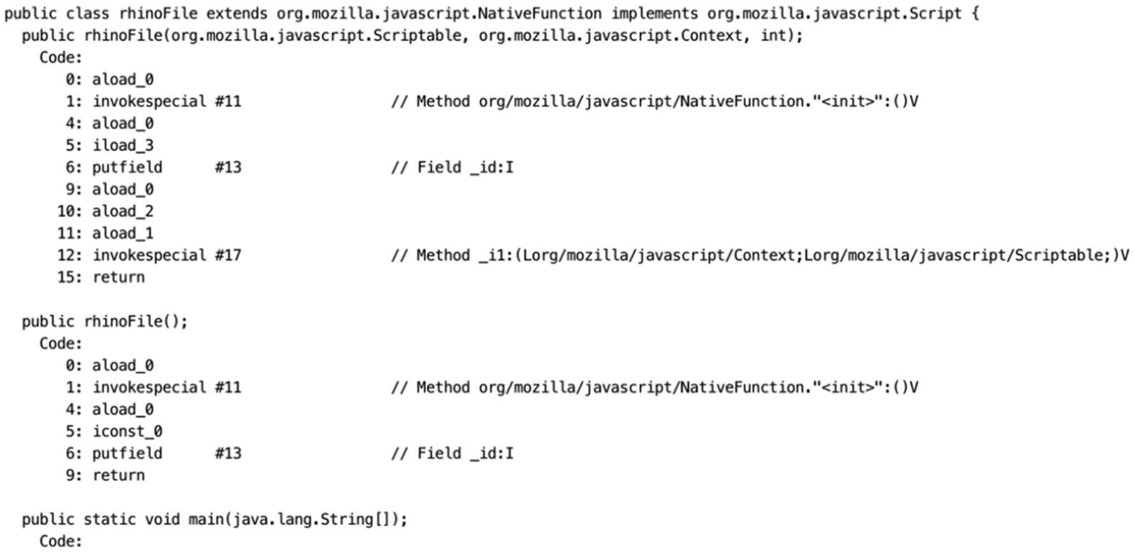


Fig. 4 Code Obfuscation techniques in Morphing Engine





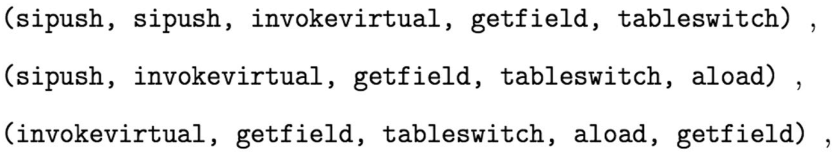
```
public class rhinoFile extends org.mozilla.javascript.NativeFunction implements org.mozilla.javascript.Script {
  public rhinoFile(org.mozilla.javascript.Scriptable, org.mozilla.javascript.Context, int);
    Code:
       0: aload_0
       1: invokespecial #11                 // Method org/mozilla/javascript/NativeFunction."<init>":()V
       4: aload_0
       5: iload_3
       6: putfield      #13                 // Field _id:I
       9: aload_0
      10: aload_2
      11: aload_1
      12: invokespecial #17                 // Method _i1:(Lorg/mozilla/javascript/Context;Lorg/mozilla/javascript/Scriptable;)V
      15: return

  public rhinoFile();
    Code:
       0: aload_0
       1: invokespecial #11                 // Method org/mozilla/javascript/NativeFunction."<init>":()V
       4: aload_0
       5: iconst_0
       6: putfield      #13                 // Field _id:I
       9: return

  public static void main(java.lang.String[]);
    Code:
```

Fig. 5 Java bytecode

Fig. 6 N-gram from the sequence file keeping N = 5



```
(sipush, sipush, invokevirtual, getfield, tableswitch) ,
(sipush, invokevirtual, getfield, tableswitch, aload) ,
(invokevirtual, getfield, tableswitch, aload, getfield) ,
```

Fig. 7 Steps involved in N-gram Feature extraction

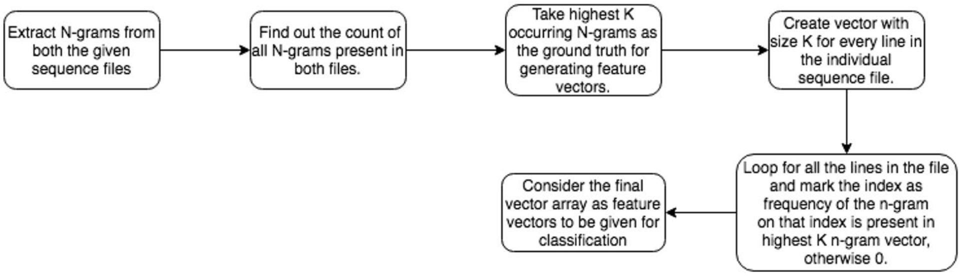
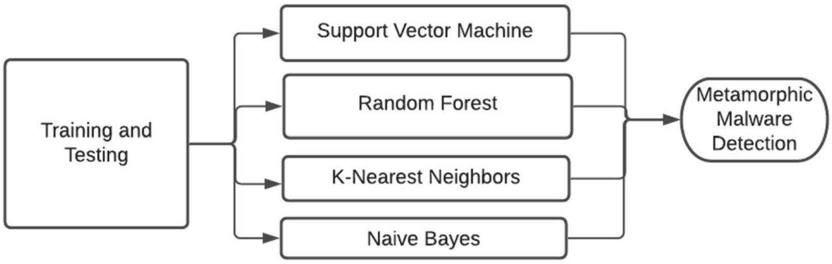

Extract N-grams from both the given sequence files
Find out the count of all N-grams present in both files.
Take highest K occurring N-grams as the ground truth for generating feature vectors.
Create vector with size K for every line in the individual sequence file.
Loop for all the lines in the file and mark the index as frequency of the n-gram on that index is present in highest K n-gram vector, otherwise 0.
Consider the final vector array as feature vectors to be given for classification

Fig. 8 Testing and training models used



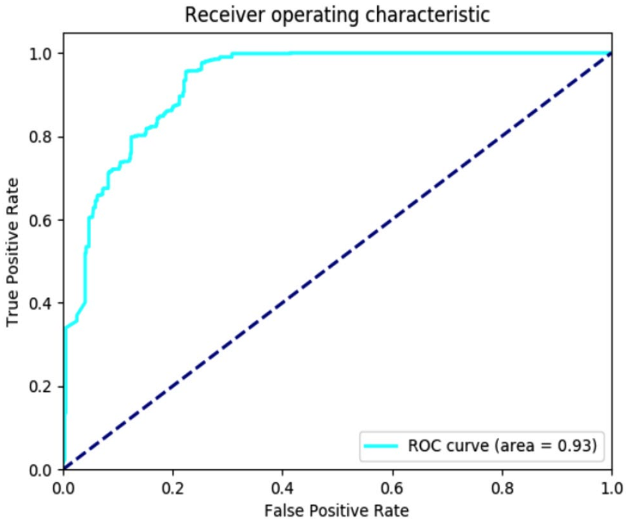


Fig. 9 Random Forest

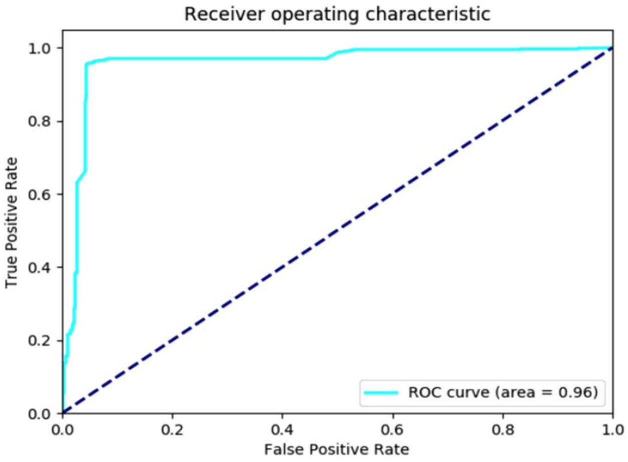


Fig. 11 Support vector machine

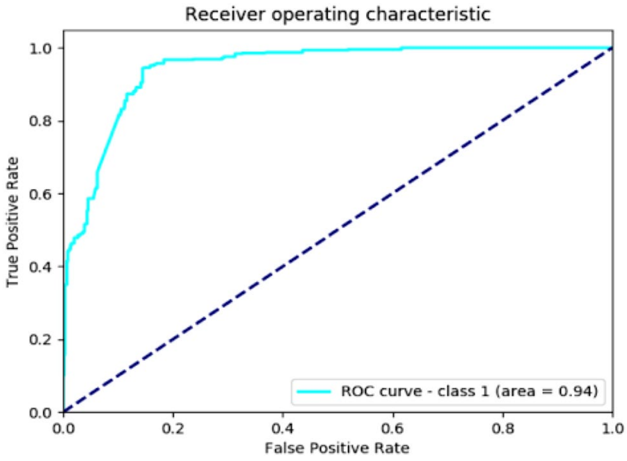


Fig. 10 K Nearest Neighbors

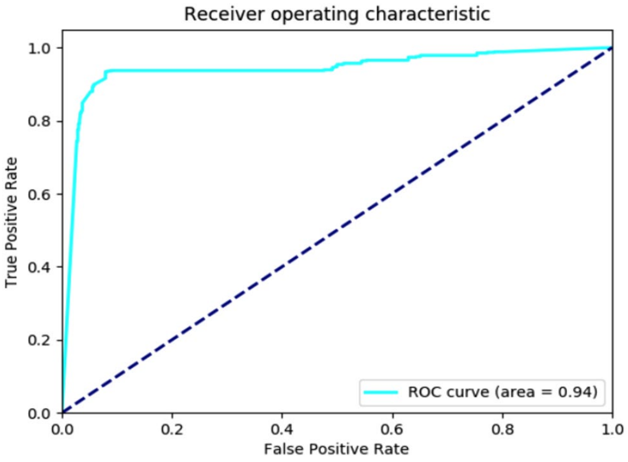


Fig. 12 Naïve Bayes

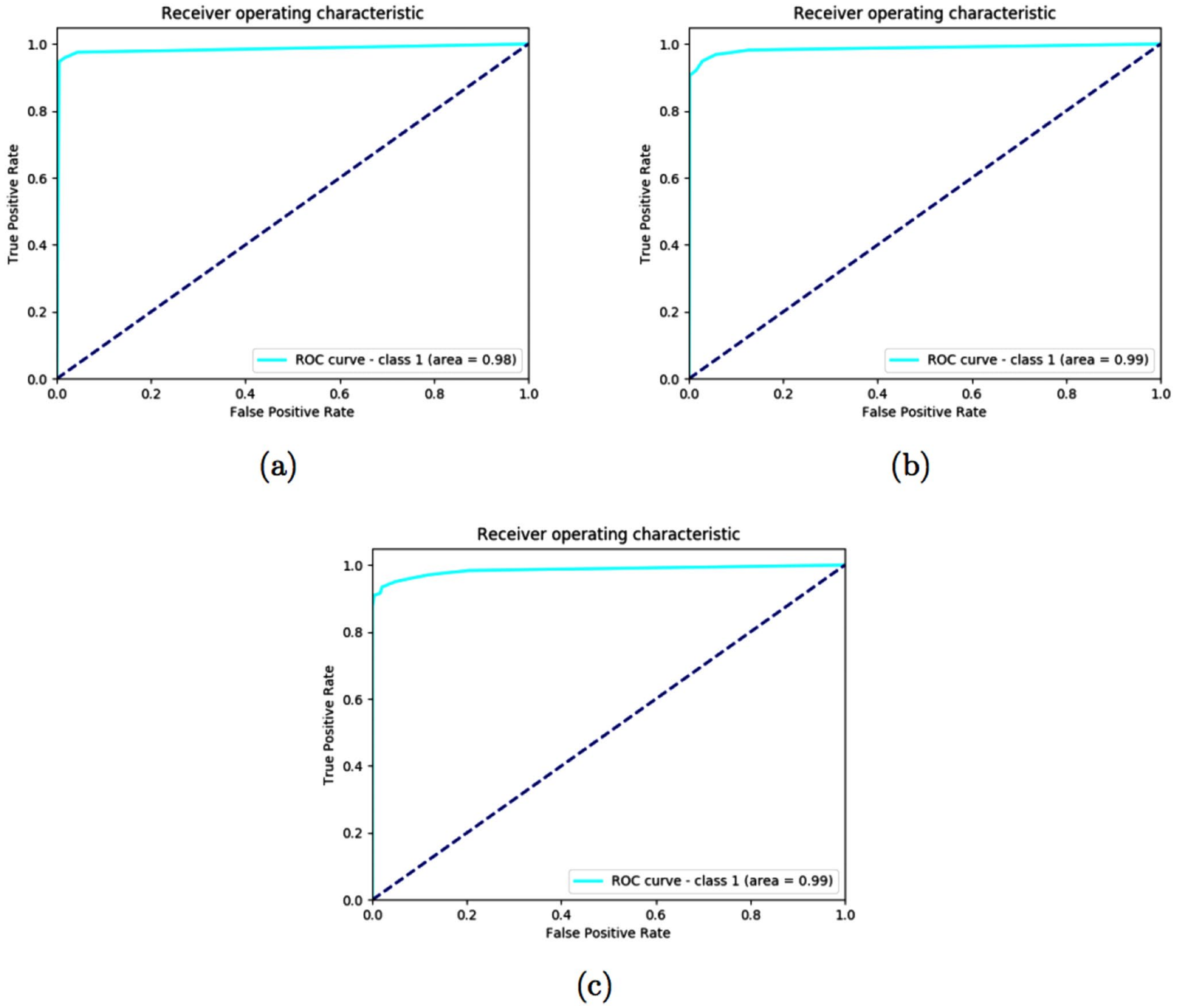


Fig. 13 KNN where $K=3$, 7, 11 respectively

Conclusion

In our research, our aim was to detect the metamorphic malwares and analyze existing techniques for the same. We have presented a framework for metamorphic malware detection using different classifiers and have done so by developing a novel framework involving java bytecode. Although the final accuracy is not higher than existing works, the model is scalable and other algorithms can be studied for better results. The main aim of the research is to imitate the morphing technique for metamorphic malwares and make a much larger and diverse dataset that ensures a better result against zero-day viruses. In future scope, we can use HMM features with specialized scoring techniques or Genetic Algorithms or even Fuzzy neural networks.

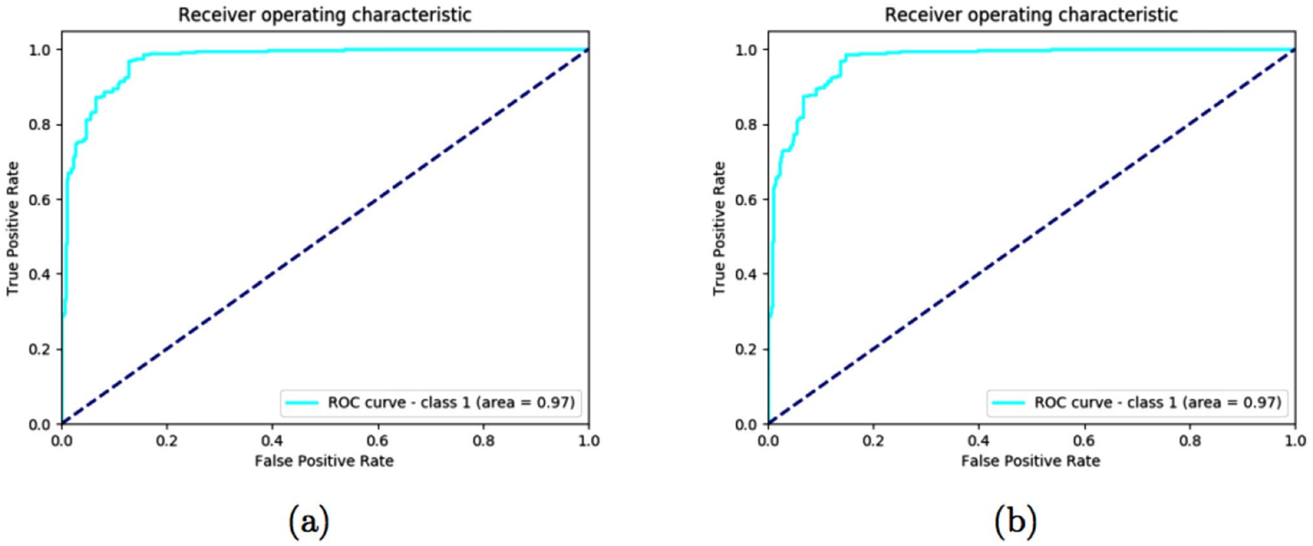


Fig. 14 Random Forest where number of Trees = 200, 300

Funding The authors did not receive support from any organization for the submitted work. The authors have no relevant financial or non-financial interests to disclose. All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Declaration

Conflict of interest The Authors declare they have no conflict of interest.

References

1. Campion M, Dalla Preda M, Giacobazzi R. Learning metamorphic malware signatures from samples. J Comput Virol Hacking Techn. 2021;17(3):167–83. https://doi.org/10.1007/s11416-021-00377-z.
2. Annachhatre C, Austin TH, Stamp M. Hidden Markov models for malware classification. J Comput Virol Hacking Techn. 2014;11(2):59–73. https://doi.org/10.1007/s11416-014-0215-x.
3. Andreopoulos WB. Malware detection with sequence- based machine learning and deep learning. Malware Anal Artif Intell Deep Learn. 2020. https://doi.org/10.1007/978-3-030-62582-52.
4. Kakisim AG, Nar M, Sogukpinar I. Metamor- phic malware identification using engine-specific patterns based on co-opcode graphs. Comput Stand Interfaces. 2020;71: 103443. https://doi.org/10.1016/j.csi.2020.103443.
5. Musale M, Austin TH, Stamp M. Hunting for metamor- phic JavaScript malware. Journal of Computer Virology and Hacking Techniques. 2014;11(2):89–102. https://doi.org/10.1007/s11416-014-0225-8.
6. Javaheri D, Lalbakhsh P, Hosseinzadeh M. A novel method for detecting future generations of targeted and metamorphic malware based on genetic algorithm. IEEE Access. 2021;9:69951–70. https://doi.org/10.1109/access.2021.3077295.
7. Souri A, Hosseini R. A state-of-the-art survey of malware detection approaches using data mining techniques. Hum-Centric Comput Inf Sci. 2018. https://doi.org/10.1186/s13673-018-0125-x.
8. Stamp Mark. A revealing introduction to hidden markov models. https://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf
9. Toderici AH, Stamp M. Chi-squared distance and meta- morphic virus detection. J Comput Virol Hacking Tech. 2012;9(1):1–14. https://doi.org/10.1007/s11416-012-0171-2.
10. Mahawer DK, Nagaraju A. Metamorphic malware detection using base malware identification approach. Secur Commun Netw. 2013;7(11):1719–33. https://doi.org/10.1002/sec.869.
11. Kancherla K, Mukkamala S. Image visualization based malware detection. IEEE Symp Comput Intell Cyber Secur (CICS). 2013. https://doi.org/10.1109/cicybs.2013.6597204.
12. Cook S. Malware statistics and facts for 2021. Comparitech. 2021. https://www.comparitech.com/antivirus/malware-statistics-facts/. Accessed 7 Oct 2022.
13. Malware Statistics Trends Report—AV-TEST. AV-Test. 2021. https://www.av-test.org/en/statistics/malware/. Accessed 7 Oct 2022
14. Nguyen VT A study of polymorphic virus detection. 2018. https://doi.org/10.13140/RG.2.2.19853.79842.
15. Ponemon Institute. The 2018 State of End- point Security Risk. 2018. https://www.ponemon.org/news-updates/news-press-releases/news/the-2018-state-of-endpoint-security-risk.html. Accessed 7 Oct 2022.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.