The Open University of Israel

Department of Mathematics and Computer Science

Improvements in Obfuscation and Detection Techniques of Malicious Code

Thesis submitted as partial fulfillment of the requirements
Towards an M.Sc. degree in Computer Science
The Open University of Israel
Computer Science Division

By:

Ishai Rosenberg

Prepared under the supervision of Prof. Ehud Gudes

November 2016

Acknowledgments

I would like to thank Prof. Gudes, my instructor, for reviewing my work, providing helpful tips and investing time in our research above and beyond the call of duty.

I would also like to thank Eitan Menachem, from Ben-Gurion University, for providing the benign third party program collection for our research.

Table of Contents

List of Figures

List of Tables

Abstract

Machine-learning has been researched as an effective method to augment today's signature-based and heuristic-based detection methods capability to cope with unknown malware. However, if an attacker gains knowledge about the machine learning algorithm, he or she can create a modified version of the malicious code, which can evade detection by the IDS.
In this thesis we create an IDS based on various classifiers, using system calls executed by the inspected code as features. We compare the detection rate of several classifiers, concluding that random forest is the best.
We then present a camouflage algorithm used to modify a malicious system call sequence to a non-detectable sequence by a decision tree classifier, while preserving the original code's functionality. This is done by adding dummy system calls to change the decision tree path of the malware to a benign path, thus misleading the classifier.
We test the stability of the camouflage for classifier's training set updates.
We also present several transformations to the classifier's input (system calls at a specific sequence index), to prevent this camouflage. We then show a modified camouflage algorithm that overcomes those transformations.
We extend our camouflage algorithm to handle random forest classifiers. Our algorithm has 100% success rate on the test set examined for both decision tree and random forest classifiers, and every input transformation variant, assuming the attacker have full knowledge about them.
Finally, we show that partial knowledge about the classifier is enough to implement a probabilistic camouflage algorithm.
Our research shows that it is not enough to provide a machine learning classifier with a large training set of benign and malicious samples to counter malware - one must also to be aware of the possibility that a machine learning algorithm would be fooled by such a camouflage algorithm to provide a wrong classification for the inspected code - and possibly try to counter such option with techniques such as the input transformation or training set updates that we've shown.
An abbreviated revision of this research was published in WISTP 2016 and NSS 2016 conferences. An extended revision would be published in the journal Concurrency and Computation: Practice and Experience.

1 Introduction

The constant rise in the complexity and number of malware entities makes improving the detection rates of intrusion detection systems (IDS) a challenging task. In this report, we refer to IDS as a tool to detect and classify malware.

Past IDS generally used two methods of malware detection:

- Signature-based detection - Searching for known patterns of data within the executable code. Malware, however, can modify itself to prevent a signature match, e.g., by using encryption. Thus, this method can be used to identify only known malware.
- Heuristic-based detection - Composed of generic signatures, including wildcards, which can identify a malware family. Thus, this method can identify only variants of known malware.

Recent IDSs are not limited to the use of signatures. Machine-learning can be used in-order to extend the IDS capabilities to cope with unknown malware by classifying training samples as benign software or malware, based on specific features which could be used afterward to classify software unseen before as malware or benign. Feature extraction can be done using static or dynamic features of the code.

However, our research shows that the arms race between security researchers and malware writers is far from over: Malware code can be transformed to render machine learning classifiers almost useless, without losing the original (malicious) functionality of the modified code. We call such a generic transformation, based on the classifier type and the features used, a *camouflage algorithm*.

In this thesis, we present a camouflage algorithm for decision tree and random forest based classifiers, whose input features are sequences of system calls executed by the code at run-time. Using system call sequences as a classifier's input was reported in [48], [18]. However, attempts to fool the classifier have also been reported in [23, 56]. The main idea presented in this thesis, assuming the enemy knows the classifier's features, is that the fooling action can be done *automatically* by a camouflage algorithm. Thus it can be applied easily to many instances of malware and creates a real problem for the IDS. This suggests that it is not enough to provide a machine learning classifier with a large DB of benign and malicious samples: One must also be aware of the possibility that a machine learning algorithm would be fooled by such a camouflage algorithm, and take defensive actions against it. Our research has three main contributions:

1. Developing an automatic algorithm to decide which system calls to add to a malware code that is classified as malicious by the IDS, to classify this code as benign by the same IDS, without losing its functionality. We also investigate the stability of the camouflage algorithm, meaning its ability to maintain its benign classification even when additional training samples are added to the training set (similar to virus definition updates of an anti-virus software). We then alleviate the assumption of full knowledge of the classifier by the attacker, showing that partial training set information might be enough.

2. Evaluating the algorithm against a large subset of malware samples, while previous work evaluated specific examples only.
3. Investigating possible transformations of the IDS input in-order to counter the camouflage algorithm and restore the true classification - as-well-as a modified camouflage algorithm to evade those transformations.

While the above contributions are shown for specific classifiers (decision tree and random forest) and for specific features as input (system call sequences), we believe the ideas are more general, and can be applied also to different classifiers with different features.

An abbreviated revision of this research was published in [41] and [42]. An extended revision would be published in the journal Concurrency and Computation: Practice and Experience, Tianjin 2016 special issue.

The rest of the thesis is structured as follows. Section 2 discusses the background and related work. Section 3 presents the problem definition and discusses the evaluation criteria for the camouflage algorithm. Section 4 describes in detail the IDS. Section 5 specifies our camouflage algorithm implementation, and section 6 presents the experimental evaluation. Section 7 shows the input transformations we've applied to counter the camouflage algorithm, and section 8 presents the relevant experimental evaluation. In section 9, we show the implementation of a camouflage algorithm to counter the input transformations. Section 10 extends the camouflage algorithm to handle random forest, dealing with both soft and hard voting. Section 11 concludes the thesis and outlines future research.

2 Background and Related Work

We divide the background and related work into three areas: discussion of IDS classifiers which are based on system calls, designing the monitoring part of such IDSs, and presentation of previously published camouflage algorithms.

2.1 Machine Learning Binary Classifier

The usage of system calls to detect abnormal software behavior has been introduced in [18]. The authors scanned traces of normal behavior (running benign programs) and build up a database of characteristic normal patterns, i.e. observed sequences of system calls. They defined normal behavior in terms of short n-grams of system calls. The authors used a small fixed size window and "slide" it over each trace, recording which calls precede the current call within the sliding window. Then they scanned new traces that might contain abnormal behavior, looking for patterns not present in the normal database: System call pairs from test traces are compared against those in the normal profile. Any system call pair (the current call and a preceding call within the current window) not present in the normal profile is called a mismatch. A system call is defined as anomalous if there are any mismatches within its window. If the number of anomalous system calls within a time frame exceeds a certain threshold - an intrusion is reported. The authors also introduced open source tools to report such anomalies ([48] and [57]).

For example, consider a mail client that is under attack by a script that exploits a buffer overrun, adds a backdoor to the password file, and spawns a new shell listening on port 80. In this case, the system call trace will probably contain a segment looking something like: *open(), write(), close(), socket(), bind(), listen(), accept(), read(), fork()*. Since it seems unlikely that the mail client would normally open a file, bind to a network socket, and fork a child in immediate succession, the above sequence would likely contain several anomalous sub-traces, and thus this attack would be easily detected.

Data mining techniques and machine learning algorithms such as Naive Bayes have also been used with the Windows platform ([46]). Other machine learning algorithms, such as decision trees, SVM, boosted trees, Bayesian networks and artificial neural networks were used and compared in-order to find the most accurate classification algorithm - with varying results (e.g.: [28] and [27] chose boosted decision trees as the most accurate method, [33] and [17] chose decision trees, etc.). The different results are affected by the exact samples in the training set and their number, the types of the features used, etc.

In [36], the authors used a feature extraction technique based on call-flow graphs (CFGs) of opcodes of the inspected code, which was introduced in [29], and then used deep learning sum-product network (SPN) model to calculate the similarity of the inspected code to the nearest malware family. While this method seems to be suited to detect the differences between different families of malware, an IDS which monitors and analyzes the system calls directly, such as ours, works on a higher abstraction level, on-which differences between malware and benign programs are

easier to detect, as the resemblance between opcode CFGs doesn't cover all cases. For instance, consider two identical programs, one is a benign back-up utility, and another is a malicious program, which has the exact same opcodes, except for a single call to a *DeleteFile()* function in the malware variant, instead-of a *CopyFile()* in the benign variant, thus deleting an important system file instead-of backing it up. Those programs would both be classified the same using an opcode CFG similarity technique, since, opcode-wise, the call to a different system call (with the same parameters) is different only in the jump address to the new function and not in the opcode. Using system calls as features, on the other hand, would help us recognize a different (and malicious) feature in the malware variant.

In-order to classify code as benign or malicious, machine-learning algorithms use distinct features as input. Types of features that have been used to classify software are either collected statically (i.e. without running the inspected code): byte-sequence (n-gram) in the inspected code (as in [28] and [27]), APIs in the Import Address Table (IAT) of executable PE headers (as in [47]), or dis-assembly of APIs in the executable (as in [45]). The features can also be collected dynamically (i.e. by running the inspected code): CPU overhead, time to execute (e.g., if a system has an installed malware driver being called whenever accessing a file or a directory to hide the malware files, then the additional code being run would cause file access to be longer than usual), memory and disk consumption (as in [33] and [34]), machine-language op-codes sequence executed (as in [44]), executed system call sequence (as in [48] and [57]) - or non-consecutive executed system call sequence (as in [43]).

A survey of system calls monitors (including analyzing system call arguments, stack trace and return value, and execution graphs) and the attacks against them were conducted in [19], stating that in-spite of their disadvantages, they are commonly used by IDS machine learning classifiers. The authors also specified similar design principles to such classifiers based IDSs and to the biological immune systems (generic, adaptable, autonomy, graduated response and diversity), with examples from their own IDS implementation.

While gathering the inspected code's features using static analysis has the advantage that it does not require running a possibly malicious code (for example, in [47] and [45]) - it has a main disadvantage: since the code isn't being run - it might not reveal its "true features" (as shown in [32]). For instance, if you inspect the APIs in the Import Address Table (IAT) of executable PE headers (as done in [47]), you would miss APIs that are being called dynamically (using *LoadLibrary()\GetProcAddress()* in Windows and *dl_open()\dl_sym()* on Unix). If you look for byte-sequence (or signatures) in the inspected code (as done in [28]), you might not be able to catch polymorphic malware, in which those signatures are either encrypted or packed and decrypted only during run-time, by a specific bootstrap code. Similar bootstrap code can be used for "legitimate", benign applications also, so detection of such code is not enough. Other limitations of static analysis and techniques to counter it appear in [32].

Thus, in our research we decided to use dynamic analysis, which reveals the true features of the code. Obviously, malware can still try to hide, e.g., by detecting if some other application (the IDS) is debugging it or otherwise monitoring its features and not operating its malicious behavior at such cases. However, doing so (correctly) is more challenging for a malware writer and in the end, in-order to operate its malicious functionality - malware must reveal its dynamic features, during run-time. However, those features can be altered in a way that would fool an IDS, as shown in the following sections.

2.2 Sandboxing Dynamic Analysis IDSs

A main issue with using a dynamic analysis IDS is the fact that it must run the inspected code, which might harm the hosting computer. In-order to prevent a damage to the host during the inspection of the code, it's common to run the code in a sandbox: a controlled environment, which isolates between the (possibly) malicious code to the rest of the system, preventing damage to the latter. Any harmful modifications done in the sandbox can usually be monitored and reverted, if needed. In-order to avoid the malicious code from detecting that it's running on a sandbox (and thus is being monitored), the sandbox should be as similar as possible to the actual system. The isolation can be done in 3 levels:

- At the application-level, meaning that the malicious code is running on the same operating system as the rest of the system, but its system calls effect only a quarantined area of the system, e.g., registry key modifications done by the code are being redirected to different keys (as done in [60]),
- On the operating-system level (e.g., VMWare Workstation), meaning the operating system is isolated (and thus damage to that operating system does not effect the host operating system) - but the processor is the same (as done in [58]).
- At the processor level, meaning all machine instruction are emulated by a software called an emulator (like QEMU, [9]).

CWSandbox ([58]) is an operating-system level sand-boxing tool. During the initialization of an inspected binary executable, CWSandbox's dll is injected into its memory to carry out API hooking. This dll intercepts all API calls and reports them to CWSandbox. The same procedure is repeated for any child or infected process. CWSandbox and the malicious code are being executed inside a virtual machine (or VM, a software implementing a completely isolated guest operating system installation within a host operating system) based on VMware Server and Windows XP as guest system. After each code analysis, the VM is reverted to a clean snapshot. An XML report of all executed system calls is being generated by this tool.

TTAnalyze ([4]) uses processor level sand-boxing to run the unknown binary together with a complete operating system in software. Thus, the malware is never executed directly on the processor. While both tools generate a system calls report, some of the major differences between TTAnalyze and CWSandbox are:

– TTAnalyze uses the open-source PC emulator QEMU rather than a virtual machine, which makes it harder for the malware to detect that it's running in a controlled environment (since all machine instruction are emulated by the software it should be transparent to the analyzed code running inside the guest OS) .
– TTAnalyze does not modify the program that it executes (e.g., through API call hooking), making it harder to detect by malicious code via code integrity checks. Instead, it is adding the inspection code in the processor level, after each basic block (a sequence of one or more instructions that ends with a jump instruction or an instruction modifying the static CPU state in a way that cannot be deduced at translation time) translated by QEMU.
– TTAnalyze monitors calls to native kernel functions (undocumented internal implementation, susceptible to changes) as well as calls to Windows API functions (documented functions that call internally to the native functions). Malware authors sometimes use the native API directly to avoid DLL dependencies or to confuse virus scanner's operating system simulations, as done in the case of CWSandbox.
– TAnalyze can perform function call injection. Function call injection allows TTanalyze to alter the execution of the program under analysis and run TTAnalyze code in its context. This ability is required in certain cases to make the analysis more precise (for example, the *CreateFile()* API could both create a new file or open an existing one, so a code that checks whether the file existed before the call should be injected before such a call in the analyzed code in-order to properly log to API call effect).

Other dynamic analysis tools are surveyed at [14].

While an emulator-based sand-boxing technique might be harder to detect - it can be done, as shown in [15], [16] and [40]. Furthermore, the significant performance degradation (up to 20 times slower, as described in [62]) makes such system vulnerable to timing attacks (as described, for example, in [26]).

Thus, in our research we use the virtual machine sandboxing approach, as mentioned in section 4.1.

2.3 Camouflage Algorithms

Adversarial machine learning is the safe adoption of machine learning techniques in adversarial settings like spam filtering or malware detection, assuming that a malicious adversary can carefully manipulate the input data exploiting specific vulnerabilities of learning algorithms to compromise the system security. The taxonomy for this concept was introduced in [2] and extended (including review of relevant examples) in [3]. Attacks against supervised machine learning algorithms have been categorized along three primary axes:

1. Attack influence - It can be *causative*, if the attack aims to introduce vulnerabilities (to be exploited at classification phase) by manipulating training data; or *exploratory*, if the attack aims to find and subsequently exploit vulnerabilities at classification phase.

2. Security violation - It can be an *integrity* violation, if it aims to get malicious samples misclassified as legitimate; or an *availability* violation, if the goal is to increase the misclassification rate of legitimate samples, making the classifier unusable (e.g., a denial of service).
3. Attack specificity - It can be *targeted*, if specific samples are considered (e.g., the adversary aims to allow a specific intrusion or he wants a given spam email to get past the filter); or *indiscriminate*.

The arms race of modifying the samples in-order to bypass the classifier was modeled using game theory terms, possibly reaching a Nash equilibrium, for a Naive Bayes classifier, in [13], assuming that the attacker has a perfect knowledge of the classifier. The problem of achieving the required data by the attacker was modeled in [30].

The purpose of the camouflage algorithm presented here is to automate the transformation of a malware code, which is being identified by the IDS, to a malicious code which should not be detected. Using the taxonomy mentioned in [2], it is an *exploratory* and *targeted* attack which aim is *integrity* violation, or *evasion*, by the terminology of [24].

Modification of the malware code features used by a decision-tree classifier based on static analysis was presented in [23]. The features used by the classification process were binary n-gram – a specific byte sequence (=sub-string of bytes) in the binary, collected via static analysis. The authors have constructed a simulation of the IDS classifier for the installed anti-virus program by submitting a diverse collection of malicious and benign binaries to the IDS classifier, via a query COM interface (*IOfficeAntiVirus*), which runs the installed anti-virus on the file-name given as an argument and returns the classification decision for this file. Then, they've manually found in the simulated classifier a feature-set similar to the attacker's code that would be classified as benign. Finally, the authors used feature insertion on the attacker's code to manually transform its feature set to the one found. The feature insertion applied was adding n-grams in a way that would change the decision tree path of the inspected code. This was done by appending the feature bytes to the end of the file, or insert them between existing sections - These bytes will be ignored by the system loader and will not be present in the process memory image when the binary is loaded. They will therefore have no effect upon the run-time behavior of the process.

In contrast, as will be explained later, we are using the system call sequence of the inspected code, as collected during the code execution (dynamic analysis) as the features of our classifier, since those features (the actual behavior of the code) present a bigger challenge to modify without affecting the code's malicious functionality. Since the decision tree features, in our case, are system calls being executed by the code at a certain time, changing the decision path of the code means we need to modify the system calls being executed by it.

Suggested ways to modify system call sequences were presented in [56]. This article deals with *mimicry attacks*, a term first coined at [55]. A mimicry attack is where an attacker is able to "develop malicious exploit code that mimics the opera-

tion of the application, staying within the confines of the model and thereby evading detection". Some mimicry attacks are based on specific exploits. For instance, in [52], a helper program to the *restore* utility program was replaced by a user-defined helper, thus unprofileable, to gain root access.

In the context of this thesis, we use a different approach: The possibility of modifying generic system call sequences of malware in-order for the IDS to classify it as benign. The authors present several ways of obfuscating the system call sequence in [56]:

1. Modifying the system calls' parameters – The authors mention that except for specific cases, the system calls' parameters are ignored by IDS. Thus, for instance, an innocuous system call: *open("/lib'libc.so", O_RDONLY)* (to load libc by an executable) looks indistinguishable (to the IDS) from the malicious call: *open("/etc/shadow", O_RDWR)* (to write to the shadow password file). Thus, benign code system call parameters could be modified to become malicious.
2. Adding semantic "no-ops" - The term "no-op" indicates a system call with no effect, or whose effect is irrelevant to the goals of the attacker. Opening a non-existent file, opening a file and then immediately closing it, reading 0 bytes from an open file descriptor, and calling *getpid()* and discarding the result are all examples of likely no-ops. Most system calls can be called with invalid arguments, making them fail although counted by the IDS: Any system call that takes a pointer, memory address, file descriptor, signal number, pid, uid, or gid can be nullified by passing invalid arguments. The authors showed that almost every system call can be no-op-ed, and thus the attacker can add any needed no-op system call to his code in-order to achieve a benign system call sequence.
3. Equivalent attacks – There is usually more than a single system call sequence to achieve the same effect. For instance, any call to *read()* on an open file descriptor can typically be replaced by a call to *mmap()* followed by a memory access. As another example, in many cases the system calls in the malicious sequence can be re-ordered. A consecutive series of *read()* or *chdir()* calls can also be replaced with a single call. Finally, most IDSs handle *fork()* by cloning the IDS and monitoring both the child and the parent application process independently. Hence, if an attacker can reach the *fork()* system call and can split the exploit sequence into two concurrent chunks (e.g., overwriting the password file and placing a backdoor in the ls program), then the attacker can call *fork()* and then execute the first chunk in the parent and the second chunk in the child, thus avoiding a malicious system call sequence.

In our work, which also uses system call sequences, we focus on the second technique only, because it's the most flexible. Using this technique, we could add no-op system calls that would modify the decision path of the inspected code in the decision tree, as desired. The main differences from our thesis:

– We have created an automatic algorithm and tested it on a large group of malware to verify that it can be applied to any malware, not only specific samples.

– We verified that the modified malicious code functions properly and evasively by executing it after its camouflage.
– We refer to partial knowledge of the attacker.

The authors mentioned several other limitations of their technique in [19] due-to the usage of code injection, which don't apply to our thesis.

[49] implemented evasion attack for PDFRATE, a random forest classifier for static analysis of malicious PDF files, by using either a mimicry attack of adding features to the malicious PDF to make it "feature-wise similar" to a benign sample, or by creating a SVM representation of the classifier and subvert it using the same method mentioned in [6].

The main difference from our work was that the mimicry attack used in the article is classifier-agnostic, while our camouflage algorithm takes advantage of the classifier type, yielding superior results.

Similarly, [51] showed an attack as a systematic process of moving an attack sequence into the IDS detection's blind region through successive attack modification and displayed specific examples of modifications of known exploits system call traces in a way that would bypass an IDS. However, the focus in these works was to demonstrate the feasibility of mimicry attacks for a few test cases only, but not to develop an automatic algorithm to generate such attacks for many samples.

Other mimicry attacks, less relevant to this thesis, can be used to forge the program counter and return address ([21]), thus able to mislead IDSs that can analyze the stack trace of the inspected code and not just the system calls and their arguments.

A similar method to ours was presented in [31]. The authors used a replacement attack, similar to our mimicry attack, to modify the malware code. They used system calls dependence graph (SCDG) with graph edit distance and Jaccard index as clustering parameters of different malware variants and used several SCDG transformations on their malware source code to move it to a different cluster. Our approach is different in the following ways:

– Our classification method is different, and handles cases which are not covered by their clustering mechanism. For instance, if we add a system call to the end of the malware code, it would always affect the SCDG graph edit distance (and might fool SCDG classifiers), but would not affect the decision tree path of the malware.
– The authors of [31] showed that their transformations can cause similar malware variants to be classified at a different cluster - but they didn't show that it can cause malware to be classified (or clustered) as a benign program, as shown in this thesis, which is the attacker's main goal. Even if we assume that their method allows benign classification, their algorithm effectiveness is lower than ours.
– Their transformations are limited to certain APIs only - and would not be effective for malware code that doesn't have these APIs. The fact that their implementation doesn't support C++ standard library and Windows Platform SDK, as used by many malware escalates this problem. In contrast, our method is applicable to those commonly used APIs.

– The authors of [31] mentioned a performance degradation of 33% due-to the added system calls. As shown in section 5.1, we got an upper bound on the performance degradation of less than 0.02% - better performance.

[50] presented an algorithm for automated mimicry attack on FSA (or overlapping graph) classifier using system call n-grams. However, this algorithm limits the malware code that can be camouflaged using it, to one that can be assembled from benign trace n-grams.

[5, 7] presented a different method: a poisoning attack. Attacker-generated samples were added to the training set of the classifier, in-order for it to subvert the classification of malware code as benign, due-to its similarity to the added samples. The problem with this method is that it requires the attacker to modify the classifier's DB, which is usually secured. Our method, which modifies the malware code and not the classifier, is more feasible to implement.

[6] showed an evasion algorithm for SVM and Neural Network classifier by following the gradient of the weighted sum of the classifier's decision function and the estimated density function of benign examples. The starting point of the gradient descent is the feature vector of the malicious sample. The starting sample is usually correctly classified as malicious; the goal is to move to the area where the classification algorithm classifies points as benign. In order to avoid moving to infeasible areas of the feature space with negative classifications, the algorithm's objective function has the second term, the density of benign examples. This ensures that the final result lies close to the region populated by real benign examples. [8, 59] added poisoning and privacy attacks against SVM classifiers.

Another technique appears in [37]. The authors used persistent interposition attacks, which can evade system-call-monitoring IDS that have perfect knowledge of the values of all system call arguments as well as their relationships, with the exception of data buffer arguments to read and write. They inject code that interposes on I/O operations performed by the victim, potentially modifying the data read or written by the victim (and stealing his data), while leaving the control-flow and other system-call arguments unmodified. This attack requires code injection vulnerability in-order to work. Thus, their method is relevant to specific applications which can utilize this specific exploit. In contrast, our method can work with any malware.

3 Problem Description

As was explained in the introduction, our main goal is to show an effective camouflage algorithm. Since this algorithm is highly dependent on the selected classifier, we deal with two separated issues:

1. Classifying the previously un-encountered inspected code, via the IDS, as benign or malicious, using its system call sequences.
2. Developing an algorithm to transform the inspected code, in-order to change the IDS inspected code classification from malicious to benign, without losing its functionality.

The general problem can be defined formally as follows:

Given the traced sequence of system calls as the array *sys_call*, where the cell: *sys_call[i]* is the i-th system call being executed by the inspected code (*sys_call[1]* is the first system call executed by the code),

Define the IDS classifier as:
classify(benign_training_set, malic_training_set, inspected_code_sys_calls), where *inspected_code_sys_calls* is the inspected code's system call array*, benign_training_set* is a set of system call arrays used to train the classifier with a known benign classification (supervised learning) and *malic_training_set* is a set of system call arrays used to train the classifier with a known malicious classification. *classify()* returns the classification of the inspected code: either benign or malicious.

Given that an inspected code generates the array: *malic_inspected_code_sys_calls,* define the camouflage algorithm as a transformation on this array, resulting with the array: *C(malic_inspected_code_sys_calls)*. The success of the camouflage algorithm is defined as follows: Given that *classify (benign_training_set, malic_training_set, malic_inspected_code_sys_calls)* = malicious, the camouflage algorithm result is:
classify(benign_training_set, malic_training_set,
C(malic_inspected_code_sys_calls)) = benign and:
malic_behavior(C(malic_inspected_code_sys_calls)) =
malic_behaviour (malic_inspected_code_sys_calls).

We demonstrate the general problem for our specific classifier's types (decision tree and random forest classifiers). However, similar ideas can be applied to other types of classifiers.

In section 5, we follow the assumption specified in [24]: The classifier's type and the features being used are known to the attacker. This is also a common assumption in cryptography (Kerckhoffs' principle). However, we also assume a stronger assumption: The classifier can be fully reconstructed by the attacker (*perfect knowledge* in [6]), by obtaining either the entire training set or the model itself. Such knowledge can be gained by reverse engineering of the IDS on the attacker's computer, without the need to gain access to the attacked host - one just needs access to the IDS. As shown in [23], an IDS decision tree can be recovered this way by exploiting public interfaces of an IDS and building the decision tree by feeding it with many benign and malicious samples and examining its classifications for them. Reconstruction attacks such as the one described in [20] for a C4.5 decision tree could

also be used for this purpose. This assumption, that the IDS classifier can be reconstructed, is common in several papers on this subject (e.g.: [5, 6, 7, 13, 19, 31, 37], etc.). In section 6.5 we would alleviate this assumption and would show that partial knowledge of the training set and no knowledge about the selected features is enough to generate a probabilistic camouflage, which performs quite well.

We further assume the attackers know the system call trace that would be produced by the malware on the inspected system. While this assumption seems quite strong since the system call trace might be affected by, e.g., files' existence and environment variables' values on the target system, it is highly unlikely, since the IDS should be generic enough to work effectively on all the clients' hosts, making system-dependent flows rare. The attacker can also get data about the targeted system without compromising it, e.g. by using tools such-as *nmap* to detect the OS[1].

3.1 Evaluating the Classification

The effectiveness of our IDS is determined by two factors (P is the probability\frequency of occurrence):

1. We would like to minimize the false negative rate of the IDS, i.e. to minimize P(*classify(benign_training_set, malic_training_set, malic_inspected_code_sys_calls)* = benign).
2. We would like to minimize the false positive rate of the IDS, i.e. to minimize P(*classify(benign_training_set, malic_training_set, benign_inspected_code_sys_calls)* = malicious).

In-order to take into account both true and false positives and negatives when comparing between different classifiers in section 6.2, we try to maximize the Matthews correlation coefficient (MCC), which is used in machine learning as a measure of the quality of binary classifications:

MCC=$\frac{TP*TN-FP*FN}{\sqrt{(TP+FP)*(TP+FN)*(TN+FP)*(TN+FN)}}$([38]).

TP - True Positive - Malware that has been classified as malicious.
FP - False Positive - Benign software that has been classified as malicious.
TN - True Negative - Benign software that has been classified as benign.
FN - False Negative - Malware that has been classified as benign.

The advantage of MCC is that other measures, such as the proportion of correct predictions (also termed accuracy), are not useful when the two classes are of very different sizes. For example, if there are more benign software than malicious in the test set, assigning a benign classification to every sample achieves a high proportion of correct predictions, but is not generally a useful classification.

3.2 Evaluating the Camouflage Algorithm

The effectiveness of the camouflage algorithm will be measured by the increased number of false negatives, i.e. we would like that:

[1] https://nmap.org/book/man-os-detection.html

P(*classify(benign_training_set, malic_training_set,C(malic_inspected_code_sys_calls))*=benign) $\geq$ P(*classify(benign_training_set, malic_training_set, malic_inspected_code_sys_calls*) =
benign). Therefore, the effectiveness of the camouflage algorithm is defined as the difference between the two probabilities (which are computed by the respective frequencies). The higher the difference between those frequencies, the more effective is the camouflage algorithm.

One would like that the camouflage algorithm will stay effective even if the classifier was modified due to new training instances. We define the camouflage to be stable if, after adding benign and malicious samples to our IDS training set and re-evaluating the classifier, the malicious code would still be classified as benign by the IDS, meaning:
classify(updated_benign_training_set, updated_malic_training_set, C(malic_inspected_code_sys_calls)) = benign. We evaluate the stability of the camouflage algorithm by the difference in the percentage of false negatives before and after the training set update.

3.3 Defending against the Camouflage Algorithm Using Input Transformations

One way to fight the camouflage algorithm is to apply transformations on the input sequences of system calls and apply a classification on the transformed sequences. The assumption is that the transformed sequences would reduce the effectiveness of the camouflage algorithm. We define an inspected system call trace transformation as *T(inspected_code_sys_calls)*. Given an inspected code, a system calls array, *inspected_code_sys_calls*, of the inspected code, and a function, *classify()*, representing the IDS classifier, we define the transformation T to be *effective* iff:

1. It would not reduce the malware detection rate, i.e.:
   P(*classify(T(benign_training_set), T(malic_training_set), T(malic_inspected_code_sys_calls))* = malicious)$\geq$
   P(*classify(benign_training_set, malic_training_set, malic_inspected_code_sys_calls)* =
   malicious)
2. It would not reduce the benign software detection rate, i.e.:
   P(*classify(T(benign_training_set), T(malic_training_set), T(benign_inspected_code_sys_calls))* = benign)$\geq$
   P(*classify(benign_training_set, malic_training_set, benign_inspected_code_sys_calls)* = benign)
3. It would reduce the camouflage algorithm effectiveness, that is, at least some of the malware samples modified by the camouflage algorithm would be detected by the classifier after the input transformation, i.e.:
   P(*classify(benign_training_set, malic_training_set, C(malic_inspected_code_sys_calls))* = benign) $\geq$
   P(*classify(T(benign_training_set), T(malic_training_set), T(C(malic_inspected_code_sys_calls)))* =benign) .

In the following sections we show the implementation of the three parts of our IDS: the implementation of the monitoring system and the classifier (section 4), the implementation of the camouflage algorithm (section 5) and the implementation of the input transformations based defense mechanism (section 7).

4 IDS Implementation

Our IDS has 3 main parts:

1. A sand-boxing mechanism that would run the inspected code inside a virtual machine for a limited amount of time.
2. Feature extraction - Recording all the system calls made by the inspected code while it's running.
3. Feature selection and classification - Feeding the system call sequence to a machine-learning binary classifier and receiving a classification for the inspected code: malicious or benign.

4.1 Sand-Boxing Mechanism

In-order to implement a dynamic analysis IDS that would sandbox the inspected code effects, we have used VMWare Workstation, a commonly used virtual machine software, where changes made by a malicious code can be reverted.

Our IDS executes the inspected code on a virtual machine with Windows XP SP3 OS without an internet connection (to prevent the possibility of infecting other machines).

Windows OS better suits our needs than Unix variants, used by available open-source tools (e.g., [48] and [57]), since most malware target it[2].

The inspected executables were run for a period of 10 seconds (and then forcefully terminated), which resulted in about 10,000 recorded system calls per executable on average (the maximum number recorded per executable was about 60,000).[3]

4.2 Feature Extraction: System Calls Recorder

The system calls recorder is the only part of our IDS code which is platform-dependent. Other than that, our IDS design is cross-platform (written in Python[4] cross-platform programming language and designed for modularity) and it could easily be modified to operate on Linux OS, using *strace()* to record system calls by the inspected code, instead-of our custom system-calls recorder.

The system calls recorder we have used for Windows records the Nt* system-calls[5]. The usage of this low layer of system calls was done in-order to prevent

[2] https://www.daniweb.com/hardware-and-software/microsoft-windows/viruses-spyware-and-other-nasties/news/310770/99-4-percent-of-malware-is-aimed-at-windows-users

[3] Tracing only the first seconds of a program execution might not detect certain malware types, like "logic bombs" that commence their malicious behavior only after the program has been running some time. However, this can be mitigated both by classifying the suspension mechanism as malicious or by tracing the code operation throughout the program execution life-time, not just when the program starts.

[4] https://www.python.org/

[5] http://undocumented.ntinternals.net/

malware from bypassing Win32API (e.g. *CreateFile()*) recording by calling those lower-level, Nt* APIs (e.g. *NtCreateFile()*). We have recorded 444 different system calls, such-as *NtClose(), NtWaitForMultipleObjects(),* etc.

There are many ways to implement such a recorder, e.g.: IAT (import address table) patching, EAT (export address table) patching, detours ([25]), Proxy\Trojan DLL or kernel hooks (e.g. SSDT or IDT). However, we decided to use debugging events, i.e., to run the inspected code via a custom-made debugger which would set breakpoints and monitor the Nt* API calls from the inspected code.[6]

This method was chose due-to the following reasons:

- It uses a documented API that is unlikely to change in following OS versions, e.g., like kernel patching was blocked by Microsoft Patch-Guard feature.
- It monitors only the inspected process, unlike, e.g., proxy dll.
- The debugger has full read and write permissions to a process it attached to and thus it's harder to fool it.

A thrall discussions of the other alternatives and why they were not chosen can be found in Progress Report #2.

4.3 Feature Selection and Classification: The Machine-Learning Classifier

We have implemented the classifier using the Python scripting language and scikit-learn[7]. This library provides a common interface for many data-mining and machine-learning algorithms. For most parts of this paper, we used the CART decision tree algorithm[8], similar to C4.5 (J48) decision tree, which was already proven to be a legitimate and even superior algorithm for malware classification (see [17] and section 6.2 for our comparison). This tree is the successor of the ID3 decision tree classifier ([39]), which was the first algorithm to use maximum entropy as a decision factor. A decision tree is invariant under scaling and various other transformations of feature values, is robust to inclusion of irrelevant features, and produces inspectable models. It is commonly used for implementing malware detection (for example in [44]).

The training set for the binary classifier contains malicious and benign executables. The malicious executables were taken from VX Heaven[9]. They were selected from the Win32 Virus type (and not, e.g., Trojan, Worm or Rootkit). This focus allowed us both to concentrate on a specific mode of action of the malicious code

[6] While, as mentioned in section 2.1, Malware can try to hide, e.g., by detecting if some other application (the IDS) is debugging it, the fact that the process is being debugged (e.g., by the API *IsDebuggerAttached()*) can be concealed by modifying the PEB, as written, for example, in: http://undocumented.ntinternals.net/source/usermode/undocumented functions/nt objects/process/peb.html

[7] http://scikit-learn.org/

[8] http://scikit-learn.org/stable/modules/tree.html

[9] http://vxheaven.org/

and to reduce the chance of infection of other computers caused by using, e.g., worm samples. Benign executables were taken from both the .\Windows\System32 folder and a collection of benign third-party programs. The number of malicious and benign samples in the set was roughly equal (521 malicious samples and 661 benign samples) to prevent the imbalanced data problem - a bias towards classification with the same value as the majority of the training samples, as presented, for example, in [63].

As features for the decision tree we used the position and the type of the system call, i.e. s*ys_call[i] = system_call_type[k]*, e.g.: *sys_call[3] = NtCreateFile*

One might argue that the position of the system call as a feature may be too specific and may creates over-fitting. In chapter 7 we relax this definition and just denote the position within a window or a region. The results of the camouflage algorithm, as can be seen in that chapter, did not change significantly. Two reasons are:

- Specific system calls are indicative of malware even in specific locations in the code, e.g. creating a remote thread using the *CreateRemoteThread()* API in the beginning of the code for code injection, or *CreateThread()* to create "logic bombs".
- Many malware writers tend to use the same bootstrap code and libraries for, e.g., code injection, privilege escalation exploits, encryption or packing, such-as Metasploit[10]. Compiler optimizations such as code blocks reordering or dead code elimination can bring different malware code variants using the same libraries to the same post-compiled system call sequence (same system calls in the same sequence position).

However the main reason for choosing those features is to ease the explanation of our algorithm in section 5. In section 7 we would use more robust features and show that our algorithm works in this case as well.

The number of available feature values was very large (about 850,000)[11]. Therefore, we performed a feature selection ([22]), selecting the 10,000 (best) features with the highest values for the χ^2 (chi-square) statistic out of the 850,000 available features, and created the decision tree based only on the selected features. If more than a single feature has the same χ^2 value, their selection order is random.

The χ^2 is a statistical hypothesis test in which the sampling distribution of the test statistic is a chi-square distribution when the null hypothesis is true. Test statistics that follow a chi-squared distribution arise from an assumption of independent normally distributed data, which is valid in many cases due to the central limit theorem. A chi-squared test can be used to reject the hypothesis that the data are independent. Here, this statistic measures dependence between stochastic variables, so a trans-

[10] https://www.metasploit.com/

[11] Note that the fact that we have 444 different system calls type being monitored and 10,000 system calls on the trace on average does not mean each of the $444^{10,000}$ combinations is being considered. A feature value is available only if it ever appeared in any part of the training set. E.g.: If *sys_call[1]* was never equal to *NtWaitForMultipleObjects* in the training set then *sys_call[1]=NtWaitForMultipleObjects* would not be a valid feature of the tree.

former based on this function "weeds out" the features that are the most likely to be independent of class and therefore irrelevant for classification.

Assume without loss of generality that if the answer is yes (i.e., *system_call[i] = system_call_type[k]*), the branch is to the right (*R* child), and if the answer is no, the branch is to the left (*L* child). An example for a simplified system-calls based decision tree is shown in Figure 1.

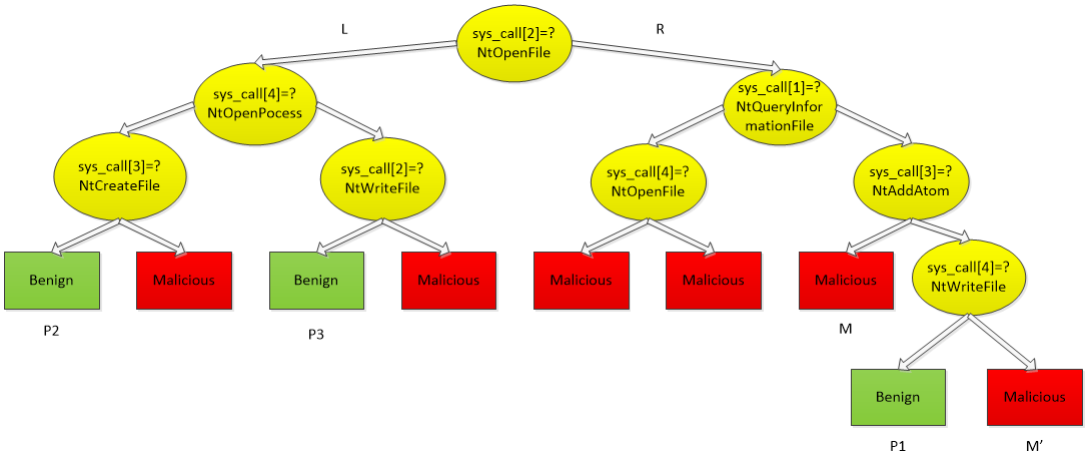


Fig. 1. A System Call-Based Decision Tree

Example 1. In this decision tree, if the inspected code trace contains:
{*sys_call[1]=NtQueryInformationFile, sys_call[2] = NtOpenFile, sys_call[3]=NtWriteFile, sys_call[4]=NtClose*},
Its path in the IDS' decision tree is: *M=RRL* (=Right-Right-Left),
since: *sys_call[2] = NtOpenFile, sys_call[1]=NtQueryInformationFile* and *sys_call[3]!=NtAddAtom*, and it would be classified as a malicious.

If the code trace contains:
{*sys_call[1]=NtQueryInformationFile, sys_call[2]=NtOpenFile, sys_call[3]=NtAddAtom, sys_call[4]=NtClose*},
the classifier would declare this code as benign,
because the decision path would be *P1=RRRL*,
since: *sys_call[2] = NtOpenFile, sys_call[1]=NtQueryInformationFile* and *sys_call[3]=NtAddAtom*, *sys_call[4]!=NtWriteFile*.

The actual decision tree generated by the training set can be found in appendix A.1.

4.3.1 Decision Tree Learning (DTL) Algorithm Our aim is to find a small tree consistent with the training examples.

The idea: (recursively) choose "most significant" attribute as root of (sub)tree:
function ID3_Decision_Tree(examples, attributes)

1. Create a root node for the tree
2. If all examples have the same target value: Return the single-node tree *root* labeled by that value
3. If no attributes were left: Return the single-node tree *root* labeled by the most common value in examples
4. else:
   a) Select the attribute A that best classifies the examples
   b) for each value v of A do:
      i. Split the tree (add a new branch to the tree corresponding to v)
      ii. If there are no examples having value v for the attribute A: Add a leaf node below this branch labeled by the most common value in examples
      iii. else: Create a subtree below this new branch

How do we choose the feature to split by? A good feature splits the examples into subsets that are (ideally) "all positive" or "all negative".

An example of two possible features appear in Figure 2.

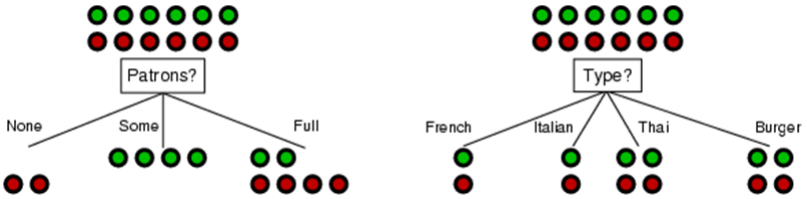


Fig. 2. Example Features to Split-by

In this case, *Patrons?* is a better choice: It minimizes the entropy and increases the uniformity of the classification of the members in the splatted groups, in comparison to the *Type?* feature.

4.3.2 Entropy To select the attribute that best classifies the examples, show in step 4.a in section 4.3.1, we'd use terms taken from Shannon's information theory:

Information Content (Entropy), ([39]):

H(V)=I(P(v_1), ... , P(v_n)) = $\sum_{i=1}^{n} -P(v_i) \lg P(v_i)$

Where $v_1 \dots v_n$ are the different possible values the random variable V.

The entropy for a training set containing p positive examples and n negative examples:

$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} \lg(\frac{p}{p+n}) - \frac{n}{p+n} \lg(\frac{n}{p+n})$

4.3.3 Information Gain (IG) A chosen attribute A divides the training set E into subsets $E_1, \dots, E_v$ according to their values for A, where A has v distinct values:

$reminder(A) = \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$

Information Gain (IG) or reduction in entropy from the attribute test is:

$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - reminder(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - \sum_{i=1}^{v} \frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$

Thus, we choose the attribute with the largest IG.

4.3.4 Gini Impurity An alternative method to choose the attribute that best classifies the examples is Gini impurity. This is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it were randomly labeled according to the distribution of labels in the subset. Gini impurity can be computed by summing the probability of each item being chosen times the probability of a mistake in categorizing that item. It reaches its minimum (zero) when all cases in the node fall into a single target category.

To compute Gini impurity for a set of items, suppose $i \in \{1,2,...,m\}$, and let f_i be the fraction of items labeled with value i in the set.

$I_G(f) = \sum_{i=1}^{m} f_i(1-f_i) = \sum_{i=1}^{m}(f_i - {f_i}^2) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} {f_i}^2 = 1 - \sum_{i=1}^{m} {f_i}^2$

This was the method used by our classifier, since it's faster to compute than IG, which contains a more expensive log calculation, with minor effect on the result classifications.

5 The Camouflage Algorithm Implementation

As was discussed above, the goal of the camouflage algorithm is to modify the sequence of system calls of the inspected code in a way that would not harm its functionality but would cause the classifier to change its classification decision from malicious to benign. This is done by finding a benign decision path (i.e., a path that starts from the tree root and ends in a leaf with benign classification) in the decision tree with the minimal edit distance ([35]) from the decision path of the malware (or the minimal Levenshtein distance between the paths' string representations) and then adding (to prevent harming the malware functionality, we are not removing or modifying) system calls to change the decision path of the modified malware code to that of the benign path. Selecting the minimal edit distance means less malware code modifications. We define the edit distance between the paths M and P as: $d(M, P)$.

Calculating the Levenshtein distance was done using the python-Levenshtein library[12]. The algorithm for finding the Levenshtein distances matrix between 2 strings appears in Appendix B and the algorithm to compute the edit operations from it appear in Appendix C).

In-order to modify the system call sequence without affecting the code's functionality, we add the required system calls with invalid parameters. This can be done for most system calls with arguments. Others can be called and ignored. For example: opening a (non-existent) file, reading (0 bytes) from a file, closing an (invalid) handle, etc.

One may claim that the IDS should consider only successful system calls to counter this method. However, such claim may not be safe because every system call may fail (e.g., a failure to open a socket due to the lack of internet connectivity, etc.). Furthermore, even if the IDS observes the function arguments, it is difficult for it to determine whether a system call is invoked with invalid parameters just to fool it, since even system calls of legitimate programs are sometimes being called with arguments that seem to be invalid, e.g.: trying to read a registry key, and creating it (only) if it doesn't already exist. Thus, such IDS might generate false classification. In addition, IDSs that verify the arguments as done in [53] tend to be much slower (4-10 times slower, as mentioned by the same authors in [54]). The conclusion is that the insertion of system calls with invalid arguments is a reasonable method to use by the camouflage algorithm.

In the basic version of our classifier, an internal node in the decision tree contains a decision condition of the form: $system\_call[i] =? system\_call\_type[k]$. As mentioned in section 4.3, we can assume without loss of generality that if the answer is yes (i.e., $system\_call[i] = system\_call\_type[k]$), the branch is to the right (R child), and if the answer is no, the branch is to the left (L child). The algorithm will add system calls that will cause the desired branching to take place, as shown in Figure 3.

[12] https://github.com/ztane/python-Levenshtein/

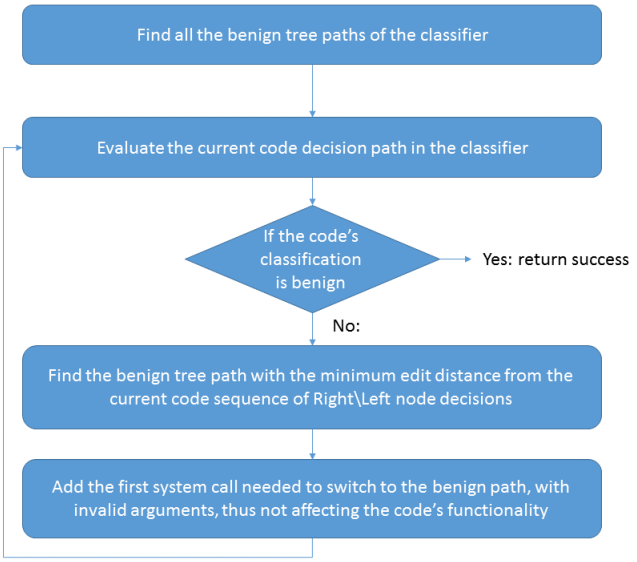


Fig. 3. The Decision Tree Camouflage Algorithm Overview

An example of a decision tree is presented in Figure 1. In this decision tree, if the malware code trace contains:
$\{sys\_call[1]$=*NtQueryInformationFile, sys_call[2]*=*NtOpenFile, sys_call[3]*=*NtAddAtom, sys_call[4]*=*NtWriteFile*$\}$
(decision path: *M'*=*RRRR*, classified as a malicious) and if the algorithm will insert as the fourth system call a system call with a different type than *NtWriteFile*, the classifier will declare this malware code as benign, since the decision path would change from *M'* to *P1*.

The general algorithm is depicted in Algorithm 1. Before explaining the details of the algorithm, lets discuss the possible edit operations when modifying a malware decision path. We will demonstrate the edit operations using two decision trees, depicted in Figure 1 and Figure 4:

1. *Substitution*: There can be two types of substitutions: Sub_L - a substitution $L \rightarrow R$ (e.g., from *P8*=*RRRL* to *P9*=*RRRR* in Figure 4) and Sub_R - a substitution $R \rightarrow L$ (e.g., from *M*=*RRL* to *P3*=*LRL* in Figure 1).
2. *Addition:* Add_R - an addition of *R* (e.g., from *M*=*RRL* to *P1*=*RRRL* in Figure 1) or Add_L - an addition of *L* (e.g., from *P7*=*RRL* to *P4*=*LRRL* in Figure 4).
3. *Deletion:* Del_L - A deletion of *L* (e.g., from *P10*=*LRL* to *P6*=*RL* in Figure 4) or Del_R - a deletion of *R* (e.g., from *P7*=*RRL* to *P6*=*RL* in Figure 4).

Since the only allowed modification is an insertion of a dummy system call, the algorithm handles the above 6 edit operations as follows:

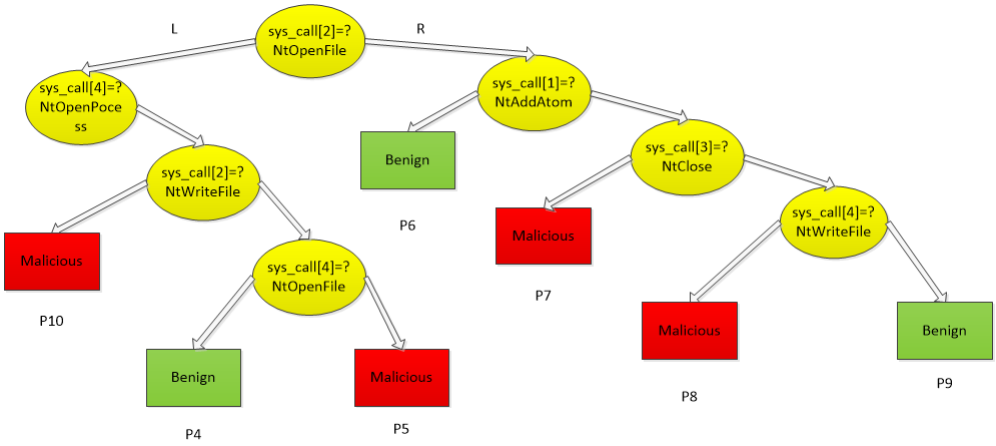


Fig. 4. A Different System Call-Based Decision Tree

– If the edit_op is Sub_L, *or* Add_R, or Del_L: Given that the condition (in the parent node of the modified\added node) is:
sys_call[i] =? *sys_call_type[k],* add *sys_call[i]=sys_call_type[k]*. Note that the equivalent of Del_L is Sub_L followed by a tree re-evaluation, since this is the only edit op allowing you to remove the L without actually deleting a system call, which might harm the code's functionality.
– If the edit_op is Sub_R, or Add_L or Del_R: Given that the condition:
sys_call[i] =? *sys_call_type[k],* add *sys_call[i]=sys_call_type[m] s.t. m != k*. The above note about deletion applies here too, with $R \leftrightarrow L$.

After each edit operation, the malware trace changes: The dummy system call addition might have affected every condition on the tree in the form of:
sys_call[j] =? *sys_call_type[k] s.t.* $j \geq i$. Therefore, we need to re-evaluate the entire decision path and find again the benign paths which are closest to it. Step 2(a) exists in-order to minimize the effects of the current edit operation on the path after re-evaluating it. Step 3 prevents an endless switch between 2 different most promising paths. *max_decision_path_count* should be chosen to be longer than the longest path in the tree.

Example 2. We demonstrate Algorithm 1 using the decision tree in Figure 1:
Given the malware code:
{*sys_call[1]=NtQueryInformationFile, sys_call[2] = NtOpenFile, sys_call[3]=NtWriteFile, sys_call[4]=NtClose*},
Its path in the IDS' decision tree is: *M=RRL* (=Right-Right-Left),
and the benign paths in the decision tree are: *P1=RRRL, P2=LLL* and *P3=LRL*,
the edit distances are *d(M, P1)=1, d(M, P2)=2, d(M, P3)=1*.
The tuples to check are: {*(M, P1), (M, P2), (M, P3)*}.
We have two paths with a minimal edit distance: *edit_sequence(M, P1)*={Add_R *(at position 3)*} and *edit_sequence(M, P3)* = {Sub_R*(at position 1)*}.

Algorithm 1 No Input Transformation System-Calls Based Decision Tree's Camouflage Algorithm

1. Given the decision tree of the IDS and a specific malware trace (i.e. its sequence of system calls as recorded) with the decision tree's path M, find all the IDS' decision tree's benign paths, *P1..Pm,* and create a list l of m tuples to check: $\{$*(M, P1)..(M, Pm)*$\}$. Set *path_count[M] = 0*
2. For each tuple *(dec_path, Pj)* in l, find the minimum edit distance between *dec_path* and *Pj*, *d(dec_path, Pj)*. Select the tuple with the minimal such edit distance and find the minimal sequence of edit operations needed to change *dec_path* to *Pj*, ordered from the root of the tree to the leaf\classification node (i.e. by position in the decision path). If l is empty: Report failure.
   a) If there is more than a single path with the same minimal edit distance, look at the first edit operation in each such path. Assuming the condition is of the form: *system_call[i]* =? *system_call_type[k],* select the path that maximizes *i.*
3. Set *path_count[des_path] += 1*. If *path_count[des_path]*$\geq$*max_decision_path_count* : Remove all tuples that contain *dec_path* from l and go to step 2.
4. Assuming the benign path to fit is *Pj*, modify the malware code based on the first edit operation in the edit sequence, as was explained above:
   a) If the edit_op is Sub_L, Add_R, or Del_L then: Add *sys_call[i]=sys_call_type[k].* Else: Add *sys_call[i]=sys_call_type[m] s.t. m != k.*
5. *system_call[i..n]* from before the modification now become *system_call[i+1..n+1].* Re-evaluate the new system call sequence and generate a new decision path *M'*.
6. If M' ends with a benign leaf: Report success. Else: Remove *(dec_path, Pj) from l,* and add all the tuples with the modified malware code $\{$*(M', P1)..(M', Pm)*$\}$ to l. Set *path_count[M'] = 0*
7. Go to step 2.

The condition for which we need to add *R in P1* is: *system_call[3] = NtAddAtom*. Thus: *i=3*.
The condition for which the edit operation applies in *P3* is:
system_call[2] = NtOpenFile. Thus: *i=2*.
Therefore, we start from *P1* and not from *P3*, since its index is larger.

In order to modify *M* to *P1*, we add: *sys_call[3] = NtAddAtom(NULL, 0, NULL)* (the edit_op is Add_R). Notice that we add the system call with invalid parameters.
The new malware code is: $\{$*sys_call[1]=NtQueryInformationFile, sys_call[2] = NtOpenFile, sys_call[3]= NtAddAtom, sys_call[4]= NtWriteFile, sys_call[5]=NtClose*$\}$.
Its decision path is *M'=RRRR*.
M' is not classified as benign – so we remove *(M, P1)*, and add all the tuples with the modified code *M'*.
Thus, we need to examine: $\{\{$*M, P2), (M, P3), (M', P1), (M', P2), (M', P3)*$\}$.
The tuple we would inspect in the next iteration is *(M', P1): d(M', P1)=1* and *i=4* (which is larger than *2* for *(M, P3)*).

The algorithm would converge after the next iteration, in which we would add *sys_call[4]!= NtWriteFile*, and the modified malware code would be classified as benign (*P1*).

The system calls insertion would ideally be done automatically, e.g, by usage of tools such-as *LLVM*, as done in [31]. As mentioned by the authors, however, such tools are currently lack support for dealing with the Windows CRT and Platform SDK API calls, as used by most Windows malware. Thus we assume that the attacker would manually insert the system calls, added by the camouflage algorithm, to the malware source code. This is demonstrated for the "Beetle" virus, in section 6.3.

While there is no guarantee that the algorithm would converge (step 3 in Algorithm 1 exists in-order to prevent an infinite loop by switching back and forth between the same paths), it did converge successfully for all the tested samples, as shown in section 6. The reason for this is the rationale behind the decision tree based on system calls: The behavior of malware (and thus the system call sequences used by it) is inherently different from that of benign software. Because-of that, and since the decision tree is trying to reduce the entropy of its nodes, the malicious and benign software are not spread uniformly at the leaf nodes of the decision tree and tend to be clustered at certain areas. Our path modifications direct the decision path to the desired cluster.

5.1 Run-Time Performance

While we hadn't measure our camouflage algorithm performance, we can estimate the effect on the code's performance by the amount of code we add. The decision tree's longest path in our classifier is less than 20 decision nodes long. Thus, we add no more than 20 system calls. If the average code flow is 10,000 system calls, we get an upper bound on the performance degradation of less than 0.02%.

In practice, most modifications were much shorter, with 1-2 additional system calls on average.

6 Experimental Evaluation

6.1 IDS Detection Rate

In-order to test the true positive detection rate of our IDS for both benign software and malware, we have used benign files collection from the Program Files folder of Windows XP SP3 and from our collection of third party benign programs and malware of Win32 Virus type, from VX Heaven's collection. The test set contained about 650 benign programs and about 500 malware. The files used to test the detection rate were different from the ones used to train the IDS in section 4.3 and thus they present code that wasn't encountered before. The malware detection rate and the benign detection rate (as computed by the definitions specified in section 3.1), were 84.3% and 88.9% respectively, as shown in the first line of Table 1.

6.2 Comparison To Other Classification Algorithms

We have chosen to attack the decision tree classifier in this thesis. However, before we quantify our results, we should verify that this classifier isn't under-performed by other classification algorithms, which would render our attack useless in practice. We've compared the effectiveness of different classification algorithms, using the same IDS, features, training set and test set. The mathematical and technical background for the other classifiers is outside the scope of this thesis, but is brought, for the reader's convenience, in Appendix D.

The evaluation criteria between the different classifiers is shown in section 3.1.

The results appear in Table 1.

Table 1. Detection Rate of the IDS by Classifier Type

Classifier Type	Malware Detection Rate (TPR)	Benign Software Detection Rate (TNR)	MCC
Decision Tree	84.3	88.9	0.76
Random Forest	86.1	89.5	0.77
K-Nearest Neighbors	89.4	86.0	0.77
Naïve Bayes (Gaussian)	87.0	54.5	0.50
Naïve Bayes (Bernoulli)	97.9	59.9	0.64
Ada-Boost	87.4	84.8	0.74
Support Vector Machine (Linear)	87.5	86.4	0.76
Support Vector Machine (RBF)	96.3	74.9	0.74

We can see that all classifiers out-performed random classification (with more than 50% detection rate for both malware and benign software).

We also see that, as expected, the Bernoulli NB classifier performs better than the Gaussian one, because the features are independent booleans.

Except for the NB classifiers (with both Gaussian and Bernoulli distributions) and the RBF-kernel based SVM, most classifiers had similar detection rates.The Random Forest classifier and k-Nearest Neighbors classifier were the best overall, taking into account both malware and benign software detection rate (by maximizing the MCC), where the k-Nearest Neighbors classifier was better in malware detection and the Random Forest classifier was better in benign software detection.

One might ask: Why is a Random Forest classifier better in classifying benign software while k-NN is better classifying malware?

As mentioned in Section 5, our assumption is that malicious software tend to have similar behavior. In a virus this might be file infetion, manifested as many file access API calls. Thus, many malware have similar features. k-NN is classifying by the nearest neighbors, which best fit when software with similar features are similarly classified. Since malware have similar features, k-NN works best for them.

Different benign software types, on the other hand, have different features, e.g., a browser would have many network API calls while a compression utility would have many file and mathematics API calls. However, different decision trees, each trained by a subset of features that fit a different type of benign software (e.g., a browser or a compression utility), would compose a Random Forest that could successfully classify benign programs.

While the Random Forest classifier and k-Nearest Neighbors classifier were the best overall, the Decision Tree classifier, used in the other sections, is almost identical in its performance; since its much easier to implement and test, we used this classifier in most of the thesis, before extending it to a random forest classifier in section 10.

6.3 Camouflage Algorithm Effectiveness

In-order to test our camouflage algorithm, we have selected all the malware samples from our test set, which were correctly classified (i.e., as malicious) by our IDS, 436 samples. We applied the camouflage algorithm on the samples. The results appear in the first line of Table 10; 84.3% of the samples were identified before applying the camouflage algorithm on their system call sequence, as shown in the first line of table 4. However, *none* of the camouflaged system call sequences of the same samples were identified by our IDS (effectiveness of 100%, by the definition in section 3, and thus the modified malware false-negative rate in Table 2 is 100%). The value of *max_decision_path_count* in section 5 was set to 30.

To test a complete "end-to-end" application of our system in real-life, we used the source code of the virus "Beetle"[13]. We compiled the source code and ran it through our IDS. The virus system call trace was classified correctly as malicious by our IDS. After using our camouflage algorithm, we received the modified system

[13]The description and the source code of this virus are available at: http://vxheaven.org/lib/vpe01.html

call sequence, which was classified as benign by our IDS. We matched the system calls in this sequence to the virus original source code, and applied the same modifications to it - and then recompiled the modified version. The modified version of the virus was then run in our IDS, and was falsely classified by it as benign. As expected, the malicious functionality of the code remained intact.

6.4 Training Set Updates

In-order to test the stability of our camouflage, we have added increments of 10 additional benign programs and 10 additional malware system call sequences to our IDS training set (or DB) - a 1.7% increment in the training set size, totaling 785 benign programs and 639 malware in the training set. This increment had a negligible effect on the detection rate of the IDS as a whole. After recalculating the decision tree, the benign detection rate changed from 88.9% to 88.7% and the malware detection rate changed from 84.3% to 86.5% for the same test set, as shown in the last line of Table 4. However, This update had a significant effect on the stability of the camouflage, by the definition given in section 3.1, as shown in Table 2. A false-negative rate of 100% means that none of the camouflaged malware system-calls traces were classified as malicious. The percentage is out of the 84.3% of malware that were successfully classified by the IDS in section 6.1.

Table 2. Percentage of Successfully Camouflaged Malware Samples When Using Updated DB

DB Increment (Percentage)	Modified Malware False-Negative Rate
0 (original DB)	100.0
1.7	74.5
6.8	67.9
11.8	63.1
13.5	50.5
16.9	44.3

We see that even less than 2% increment in the training set causes a 75% stability rate. The reason is that even a small increment cause a re-calculation of the entire decision tree, where the distinguishing features might be different from the original decision tree, rendering the camouflage useless. Furthermore, we see that the larger the increase in the training set - the lower the stability rate becomes (down to 44%, when the training set size is being increased by 17%).

This means that training set updates can decrease the camouflage algorithm effectiveness.

However, frequent classifier updates isn't a big concern for an attacker: the classifier size (300-400MB in our case - not including sandbox update of GBs of data), makes a full replacement of the classifier expensive, and would cause maintainability issues. To mitigate that, replacing it infrequently would allow an attacker to

modify the malware code to fit the new classifier. Another possibility is using incremental decision trees, updating the classifier in the client's system instead-of replacing the entire classifier. The camouflage against incremental CART algorithms, as mentioned in [12], is more stable by design since in-order to reduce the computation burden by invoking the CART procedure repeatedly, the decision tree usually remains the same, or have a new split close to the original tree leaves. The camouflage would become infective only if the malware code flaw was modified, which is less likely in this case. Since scikit-learn doesn't have an incremental decision trees implementation, we have tried to implement the CART extension in Python on our own. However, due-to the programming language memory limitations - we weren't able to create incremental decision trees with a training set size of more than 150 samples - about 10% the size of our scikit-learn classifiers' training set. Since implementing such trees in a different language requires a complicated interop mechanism that is beyond the limits of this thesis, the numerical results of testing our camouflage against incremental CART decision trees would be presented in future research.

However, even if we take the results as-is, a 50% success rate for the camouflage algorithm is not something which is tolerable by an IDS - and the system would be considered unsafe against the camouflage algorithm variants even in such case.

6.5 Partial Knowledge of the IDS

So far, we have assumed that the attacker has full knowledge of the classifier type, the training set used to create it, and its features, in-order to generate the exact same classifier and use it to camouflage the malicious code. We can alleviate this assumption: If the attacker can gain partial knowledge about the training set, he can construct the simulated classifier using *only* the training set he knows about and use it in Algorithm 1. Such partial knowledge is easy to gather, e.g., using the VirusTotal[14] samples closest to the IDS release date, which are very likely to be used as part of the IDS training set. This makes the implementation of this variant more feasible.

6.5.1 Partial Training Set Knowledge We have trained the attacker classifier using a part of the training set which is used by the IDS classifier, as mentioned in section 6. We then camouflaged the entire test set using Algorithm 1, based on the attacker partial knowledge based classifier. The success rate of the camouflage is shown in Table 3.

Notice that the training set updates, mentioned in the previous section, also reflect partial knowledge of the training set: full knowledge of the original training set, but not about the updates. The numbers in both cases (Tables 2 and 3) are similar: about 50% camouflage algorithm effectiveness for knowledge of 85% of the training set.

[14] https://www.virustotal.com/

Table 3. Camouflage Algorithm Effectiveness Using Partial DB

Percentage of Training Set known to the Attacker	Modified Malware False-Negative Rate
100 (original DB)	64.0
86.4	56.6
77.7	31.3
69.1	25.4

6.5.2 Full Training Set Knowledge, Lack of Features' Knowledge As mentioned in section 4.3, we select only the most statistically-significant 10,000 features. If more than a single feature has the same χ^2 value, their selection order is random. In this case, a different classifier might be generated by the attacker from the same training set. The same is correct for Gini impurity, during the construction of the decision tree out-of those selected features. Therefore, we tested a case of full knowledge about the training set, but not about the features being selected, in case of chi-square equality during feature selection and Gini impurity equality during decision tree building. As can be seen in the first line of Table 3, in this case the camouflage is 64% effective. However, as the training set grows, the probability of getting the same classifier increases, since the features would tend to have different chi-square and Gini impurity values, and the effect lacking feature knowledge would be reduced.

From all experiments, it is clear that the camouflage algorithm is useful to an attacker - even with partial knowledge of the classifier.

7 Countering the Basic Camouflage: Input Transformation

As mentioned in the previous sections, the input to our IDS is a sequence of *(system_call_type, position*) tuples, represented as an array of system calls. The basic form of decision tree node condition is: *system_call[i]=?system_call_type[k]*. However, using this kind of input makes the IDS classification fragile: It's enough that we add a single system call in the middle of the sequence or switch the positions of two system calls, to change the entire decision path.

Therefore, we want to transform the input data (the system call array) in a way that would make a modification of the inspected code harder to impact the modified code's decision tree path, thus countering the camouflage algorithm.

7.1 Section-based transformations

In-order to define those transformations, we first divide the system call sequence to small sections of consecutive system calls. Each system calls section would have a fixed length, *m*.
Thus, *section[i]*=$\{$*sys_call[(i-1)*m+1],..,sys_call[i*m]*$\}$.

In an *order-preserving without duplicates removal section-based transformation*, we define the discrete values of the the decision nodes in the tree to be:

section[i] =? *sys_call[(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m]*.
Notice that when *m*=1 this is equivalent to no transformation.

Example 3. If the inspected code has 6 system calls:

sys_call[1]=NtQueryInformationFile, sys_call[2]=NtOpenFile, sys_call[3]=NtYieldExecution, sys_call[4]=NtWriteFile, sys_call[5]=NtWriteFile, sys_call[6]=NtClose and *m=3*
then
section[1] = NtQueryInformationFile, NtOpenFile, NtYieldExecution and *section[2]=NtWriteFile, NtWriteFile, NtClose*

However, this transformation is more specific than the basic model - so it would be easier to fool - and thus we didn't use it.

This changes when adding *duplicates removal*: If there is more than a single system call of the same type in a section - only the first instance (which represent all other instances) appears in the section. This transformation prevents the possibility of splitting a system call into separate system calls: e.g. two *NtWriteFile()* calls, each writing 100 bytes, instead of a single call writing 200 bytes. Therefore, this was the first transformation we used.

Example 4. In the example shown in the former transformation:

section[1] = NtQueryInformationFile, NtOpenFile, NtYieldExecution and
section[2]= NtWriteFile, NtClose.

The second section-based transformation we examined is *non-order-preserving without duplicates removal*. This transformation is identical to the *order-preserving without duplicates removal transformation*, except for the fact that the system calls

in each section are ordered in a predetermined manner (lexicographically), regardless of their order of appearance in the trace. Using this transformation decreases the probability of affecting the decision tree path by switching the places of two arbitrary system calls. Only the switching of two system calls from different sections might affect the decision path.

Example 5. In the example shown in the former transformation:
section[1] = NtOpenFile, NtQueryInformationFile, NtYieldExecution and *section[2]=NtClose, NtWriteFile, NtWriteFile*.

In this transformation, the probability of affecting the decision tree path by switching the places of two arbitrary system calls is $\frac{n-m}{n}$ times the probability of the *non-order-preserving without duplicates removal* transformation, since switching of two system calls affect the node condition only if the two system calls are in different sections.

The last transformation we considered is *non-order-preserving with duplicates removal*. This transformation is identical to the former, except for the fact that only one instance (which represents all other instances) would appear in the section if there is more than a single system call of the same type in a section. This transformation handles both system calls switching and splitting. Notice that this transformation makes a section similar to a set of system calls: Each value can appear at most once, without position significance.

Example 6. In the example shown in the former transformation:
section[1] = NtOpenFile, NtQueryInformationFile, NtYieldExecution and section[2]=NtClose, NtWriteFile.

7.2 Histogram-based transformations

The histogram-based transformations switch the keys and values of the section-based transformations, that is, for each system call we enumerate all the sections in which it appears:

The first *histogram-based transformation* is *without duplicates removal*. In this transformation, we concatenates all the sections in-which a system call type appear.

Example 7. In the example from section 7.1:
NtQueryInformationFile=section[1], NtOpenFile=section[1], NtYieldExecution=section[1], NtWriteFile=section[2], section[2] and: NtClose=section[2].

The second *histogram-based transformation* is *with duplicates removal*.

Example 8. In the example from the last transformation:
NtQueryInformationFile=section[1], NtOpenFile=section[1], NtYieldExecution=section[1], NtWriteFile=section[2], and: NtClose=section[2].

8 Experimental evaluation of the transformed IDS model

8.1 IDS detection rate - Decision trees

In-order to test the true positive detection rate of our modified IDS, we used the same test set used for the basic model.

A section size of *m=10* was chosen.

The detection rate, computed by the definitions specified in section 3.1, appear in Table 4.

Table 4. Detection Rate of the Decision Tree Classifier by Input Transformation Type

Input Type	Malware Detection Rate	Benign Software Detection Rate
No transformation - original DB	84.3	88.9
Section-based, non order-preserving, without duplicates removal	87.4	90.7
Section-based, non order-preserving, with duplicates removal	86.5	88.1
Section-based, order-preserving, with duplicates removal	87.6	91.3
Histogram-based, without duplicates removal	72.5	94.5
Histogram-based, with duplicates removal	72.3	93.9
No transformation - updated DB	86.5	88.7

As can be seen from this table, section-based transformations are effective, by the definition in section 3, while histogram-based transformation aren't. The best input-transformed classifier, *order-preserving, with duplicates removal*, appear in Appendix A.2.

8.2 IDS detection rate - Other classifiers

In section 6.2, we have presented different classifiers detection rates. The effect of the input transformations mentioned in section 7 on their detection rate appear in Tables 5-9.

We can see that in-general, histogram-based outperformed section-based input transformations by the criteria provided in section 3.1. We also see that different classifiers have different better-suited input transformation. For instance, *section-based, non order-preserving, without duplicates removal* is better suited for a Linear SVM classifier while *section-based, order-preserving, with duplicates removal* transformation is better suited for random forest classifier.

Table 5. Detection Rate of the Naïve Bayes (Gaussian) Classifier by Input Transformation Type

Input Type	Malware Detection Rate	Benign Software Detection Rate
No transformation	87.0	54.5
Section-based, non order-preserving, without duplicates removal	91.3	72.5
Section-based, non order-preserving, with duplicates removal	88.8	74.3
Section-based, order-preserving, with duplicates removal	87.0	76.1
Histogram-based, without duplicates removal	77.2	91.1
Histogram-based, with duplicates removal	77.0	91.5

Table 6. Detection Rate of the Linear SVM Classifier by Input Transformation Type

Input Type	Malware Detection Rate	Benign Software Detection Rate
No transformation	86.5	86.4
Section-based, non order-preserving, without duplicates removal	88.0	90.0
Section-based, non order-preserving, with duplicates removal	84.1	88.7
Section-based, order-preserving, with duplicates removal	87.0	88.7
Histogram-based, without duplicates removal	65.4	97.2
Histogram-based, with duplicates removal	64.4	97.4

Table 7. Detection Rate of the AdaBoost Classifier by Input Transformation Type

Input Type	Malware Detection Rate	Benign Software Detection Rate
No transformation	87.4	84.8
Section-based, non order-preserving, without duplicates removal	86.9	87.5
Section-based, non order-preserving, with duplicates removal	86.9	89.3
Section-based, order-preserving, with duplicates removal	83.0	91.7
Histogram-based, without duplicates removal	68.3	92.7
Histogram-based, with duplicates removal	68.5	91.7

Table 8. Detection Rate of the Random Forest Classifier by Input Transformation Type

Input Type	Malware Detection Rate	Benign Software Detection Rate
No transformation	86.1	89.5
Section-based, non order-preserving, without duplicates removal	82.8	91.9
Section-based, non order-preserving, with duplicates removal	82.6	91.9
Section-based, order-preserving, with duplicates removal	86.3	91.5
Histogram-based, without duplicates removal	59.2	98.4
Histogram-based, with duplicates removal	60.9	97.0

Table 9. Detection Rate of the k-NN Classifier by Input Transformation Type

Input Type	Malware Detection Rate	Benign Software Detection Rate
No transformation	89.4	86.0
Section-based, non order-preserving, without duplicates removal	78.9	93.7
Section-based, non order-preserving, with duplicates removal	81.6	93.1
Section-based, order-preserving, with duplicates removal	80.3	92.1
Histogram-based, without duplicates removal	42.2	98.6
Histogram-based, with duplicates removal	40.2	98.4

8.3 Camouflage algorithm - Decision trees

In-order to test our camouflage algorithm stability vs. the modified IDS, we have used the same malware samples which were used before. That-is, we used the camouflage algorithm shown in Algorithm 1 to modify their system call traces. Then we applied the input transformation on the modified system call traces - we then fed them to the input-transformed IDS variant. The results appear in Table 10.

We see that each input transformation decreases the effectiveness of the camouflage generated by Algorithm 1 dramatically. This makes sense, since Algorithm 1 modifies single system calls, not entire sections or system calls frequencies, as detected by the input transformations mentioned in this section. In the next section we modify the camouflage algorithm, to deal with input transformations and remain effective.

Table 10. Percentage of Successfully Camouflaged Malware Samples When Using Input Transformations

Input Type	Modified Malware False-Negative Rate
No transformation (original DB)	100.0
Section-based, non order-preserving, without duplicates removal	18.8
Section-based, non order-preserving, with duplicates removal	17.2
Section-based, order-preserving, with duplicates removal	17.4
Histogram-based transformation, without duplicates removal	24.8
Histogram-based transformation, with duplicates removal	21.1

9 Countering the Input Transformations with Custom-Fit Camouflage Algorithm

One might argue that camouflaging a system call trace in our basic IDS (without the transformations suggested in section 7) is an easy task. One needs to add only a single system call at the beginning to change all following system calls positions, thus affecting the decision path in the tree. Can we apply our camouflage algorithm on our section-based IDS with the same effectiveness?

In-order to fit our camouflage algorithm to section-based transformations, we have slightly modified Algorithm 1: In each iteration we added an entire system calls section (i.e., m consecutive system calls), instead of a single system call. This is done in step 4: If the edit_op is Sub_L, Add_R, or Del_L then we add the same section. Else, we add a section with a different first system call (thus, we add a different section). The section is added with the same transformation type as the IDS: If the input transformation for the decision tree was *order preserving*, so would the section added. If the input transformation included *duplicates removal*, so would the added section - and vice-versa.

The modified version is shown in Algorithm 2.

Notice that when the section size, m, is 1 - Algorithms 1 and 2 are identical.

We have applied this algorithm on all section-based transformations described in section 7.1, both order preserving and not, and both with and without duplicate removal.

Like Algorithm 1, there is no guarantee that the algorithm would converge. However, all 436 modified section traces were classified as benign by our IDS, with all input transformations, i.e., camouflage algorithm effectiveness of 100% (as can be seen in the first line of Table 11). This is due-to the same rationale mentioned in section 5.

A camouflage algorithm for histogram-based input transformation, as those mentioned in section 7.2, wasn't developed, since those transformations aren't effective by the definition in section 3.2, so they would less likely to be used. In this case, the back-tracking from the modified path to the modified malware source is more challenging. We might discuss this algorithm in our future research.

Algorithm 2 Section-Based Input Transformed System-Calls Based Decision Tree's Camouflage Algorithm

1. Given the decision tree of the IDS and a specific malware trace (i.e. its sequence of system calls as recorded) with the decision tree's path M, find all the IDS' decision tree's benign paths, *P1..Pm,* and create a list l of m tuples to check: $\{$*(M, P1)..(M, Pm)*$\}$. Set *path_count[M] = 0*
2. For each tuple *(dec_path, Pj)* in l, find the minimum edit distance between *dec_path* and *Pj*, *d(dec_path, Pj)*. Select the tuple with the minimal such edit distance and find the minimal sequence of edit operations needed to change *dec_path* to *Pj*, ordered from the root of the tree to the leaf\classification node (i.e. by position in the decision path). If l is empty: Report failure.
   a) If there is more than a single path with the same minimal edit distance, look at the first edit operation in each such path. Assuming the condition is: *section[i]* =? *sys_call[(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m],* select the path that maximizes *i.*
3. Set *path_count[des_path] += 1*. If *path_count[des_path]*$\geq$*max_decision_path_count* : Remove all tuples that contain *dec_path* from l and go to step 2.
4. Assuming the benign path to fit is *Pj*, modify the malware code based on the first edit operation in the edit sequence, as was explained above:
   a) If the edit_op is Sub_L, Add_R, or Del_L then: Add *section[i] = sys_call[(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m].* Else: Add *section[i]=sys_call'[(i-1)*m+1], sys_call[(i-1)*m+2],.., sys_call[i*m] s.t. sys_call'[(i-1)*m+1] != sys_call[(i-1)*m+1].*
5. *section[i..n]* from before the modification now become *section[i+1..n+1]*. Re-evaluate the new system call sequence and generate a new decision path *M'*.
6. If M' ends with a benign leaf: Report success. Else: Remove *(dec_path, Pj) from l,* and add all the tuples with the modified malware code $\{$*(M', P1)..(M', Pm)*$\}$ to l. Set *path_count[M'] = 0*
7. Go to step 2.

9.1 Training Set Updates

In-order to test the stability of our transformed camouflage, we have used the same training set and test set we used in section 6.3. We have applied the input transformation on both, and then run the transformed camouflage algorithm, which was done by the original training set (not the updated one), on the test set. Following that, we fed it to the input-transformed IDS variant after the training set (DB) update and re-calculating the decision tree. Like in section 6.3, we see there is no significant effect on the detection rate. However, this update had a significant effect on the stability of the camouflage by the definition given in section 3.1, as shown in Table 11.

We see that for all input transformations the larger the increase in the training set - the lower the stability rate becomes. We can also see that the stability of the camouflage algorithm reduces more drastically when using the input transformed IDSs' than when using the non input-transformed IDS, when the training set size is being updated. This makes sense, because the input transformation make the IDS

Table 11. Percentage of Successfully Transformed Camouflaged Malware Samples When Using Updated DB

DB Increment (Percentage)	No transformation	Non order-preserving, without duplicates removal	Non order-preserving, with duplicates removal	Order-preserving, with duplicates removal
0 (original DB)	100.0	100.0	100.0	100.0
6.8	67.9	56.9	81.2	40.8
11.8	63.1	55.5	30.7	52.3
13.5	50.5	48.3	26.2	53.2
16.9	44.3	40.4	24.8	47.2

more effective and less sensitive to modifications in the malware (including camouflage). Therefore a training set update should make the camouflage easier to detect - and thus less stable.

10 Random Forest Camouflage Algorithm

In section 5 we've devised a camouflage algorithm to counter a decision tree classifier. However, as mentioned in section 6.2, the classifier with the best performance was random forest. This algorithm is known to be one of the best-of-breed not only in our tests, but in other cases - as mentioned in [28] and [27]. As mentioned in appendix D.1, a random forest is actually a collection of decision trees. Thus, if we extend the same assumptions made in section 3, that-is: we know all the trees in the random forest, we can extend the algorithm mentioned in section 5 to handle random forests. The general algorithm is depicted as Algorithm 3.

Algorithm 3 No Input Transformation System-Calls Based Random Forest's Camouflage Algorithm

1. Given the random forest of the IDS, containing the decision trees *RF*={*RF_DT[1],..RF_DT[n]*} and a specific malware trace (i.e. its sequence of system calls as recorded) with the decision tree's path M, set *curr_tree_ind = 0*, *curr_trace* = M.
2. Calculate *curr_non_bypassed_trees* for *curr_trace* by feeding it to all the trees in *RF*. Set *curr_bypassed_trees_num* =|*RF*|-|*curr_non_bypassed_trees*|.
3. If *curr_bypassed_trees_num*>|*RF*|/2, return *curr_trace.*
4. Set *curr_tree_ind* = *curr_tree_ind* + *1*
5. If *curr_tree_ind*>|*RF*|*:*
   a) If *curr_non_bypassed_trees* is empty, return Failure: Else: *curr_tree_ind* = *curr_non_bypassed_trees.pop()*
6. If *RF_DT[curr_tree_ind]* classifies *curr_trace* as benign, go to step 4.
7. Operate Algorithm 1 on *RF_DT[curr_tree_ind]* and *curr_trace* to generate *modified_trace.*
8. If Algorithm 1 returned Failure, go to step 4.
9. Calculate *new_non_bypassed_trees*, the list of trees which classify *modified_trace* as malicious. Set *new_bypassed_trees_num* =|*RF*|-|*new_non_bypassed_trees*|.
10. If *curr_bypassed_trees_num*>*new_bypassed_trees_num*, go to step 4.
11. Set: *curr_trace=modified_trace, curr_bypassed_trees=new_bypassed_trees*, *curr_bypassed_trees_num=new_bypassed_trees_num.* Go to step 2.

The rationale of the algorithm is simple: Since all decision trees in *RF* actually represents parts of the same code flow, we can modify each of them in turn, using Algorithm 1, until we can fool the majority of them, thus fooling the entire random forest (step 3). In-order to finish as-fast-as-possible, the algorithm uses a back-tracking forest state, where *curr_trace* is the current (possibly already modified to fool other trees) system call trace of our malware, *curr_tree_ind* is the index of the decision tree we currently trying to fool (*i* to fool *RF_DT[i]*), *curr_non_bypassed_trees* is a queue of trees which still classify *curr_trace* as malicious, and *curr_bypassed_trees_num* is the number of trees in *RF* currently being fooled by *curr_trace* (steps 1 and 2). Since sometimes modifying the code can affect a tree we've already successfully

fooled, we skip every step which makes our state worse by reducing the amount of trees we've fooled so far - we advance only if the number of fooled trees doesn't decrease (step 11). Otherwise we implement a back-tracking strategy (step 10). We don't have to fool all the trees in the random forest - only the majority of them (step 3). We don't try to modify the code for trees which already classify our code as benign (step 6). If we don't succeed fooling a specific tree, we try to fool the next trees (step 8). Once we've passed all the trees and still don't have majority of fooled trees we try again the trees we've skipped, hoping that they would succeed now. If all trees have been tried but we still don't have majority we return Failure (step 5.a).

Notice that when the number of decision trees in the forest, n, is 1, algorithms 1 and 3 are identical.

We have applied this algorithm on all the malware code that were detected by the random forest: 445 different samples.

Like Algorithm 1 and 2, there is no guarantee that the algorithm would converge. However, all 445 modified section traces were classified as benign by our IDS, i.e., camouflage algorithm effectiveness of 100%. This is due to the same rationale mentioned in section 5. This was also the case when adding input transformation and used Algorithm 2 instead of Algorithm 1 in steps 7 and 8 of Algorithm 3.

10.1 Handling Soft Voting

In contrast to the CART trees described in [10], scikit-learn combines classifiers by averaging their probabilistic prediction (soft voting), instead of letting each classifier vote for a single class[15]. This creates a challenge to our algorithm, which handles majority vote (hard voting). Thus, we have modified our algorithm: in step 3 of Algorithm 3, we changed the condition: We're returning the current trace (success) only if the random forest as a whole classifies *curr_trace* as benign. This is true also to the back-tracking, which now conditions the random forest benign probability and should be bigger than before (step 10).

We also tested our hard voting algorithm (Algorithm 3 without modifications) against the soft voting random forest IDS, for all samples of all input transformations mentioned before: 1 sample out of 1746 wasn't successfully camouflaged, 99.94% success rate. When we used the soft voting algorithm, we got a 100% success rate for all input transformation variants.

10.2 Shortest Tree Edit Distance Optimization

In Algorithm 3, we go over the decision trees sequentially, as shown in step 4. This might not be an optimized way to iterate the trees: If two trees are similar in their nodes, it makes more sense to iterate over them one-after-the-other, under the assumption that if two trees are similar, only a small modification would be needed to fool both.

[15] http://scikit-learn.org/stable/modules/ensemble.html#random-forests

One way to determine the similarity between trees is the Zhang-Sasha algorithm for tree edit distance ([64]). If, instead of iterating the trees sequentially, we would first calculate the edit distance between the different trees and iterate them by edit distance from each other - we might finish with less edit operations.

In-order to examine this, we've applied the Zhang-Sasha algorithm, written in Python[16], on the random-forest trees. Unfortunately, the decision trees in the random forest were very distinct from each other (an average edit distance of more than 200) - and the time it took to calculate the edit distance was significant, compared to the negligible performance boost gained due-to changing the iteration order.

[16] https://github.com/timtadh/zhang-shasha

11 Conclusions

While the usage of machine learning algorithms to detect malware is a commonly researched subject, the research of camouflage techniques of malware, to mislead such algorithms, has just began scratching the surface. In this thesis, we have shown that malware code which has been identified by specific machine learning classifier (decision tree and random forest) can be camouflaged in-order to be falsely classified as benign. We have done so by modifying the actual code being executed, without harming its malicious functionality. We have implemented such a camouflage algorithm not only for a decision tree classifier, but also for one of the best-of-breed classifiers: Random Forest. We also applied a defense mechanism to the first camouflage algorithm, called input transformation, making our IDS more robust than basing it on (position, type) pairs of system calls. We also showed that it can also be evaded.

This suggests that the malware - anti-malware war is ongoing: It is not enough to use a machine learning classifier with a large training set of benign and malicious samples to detect malware: one must also be aware of the possibility that a machine learning algorithm would be fooled by this camouflage algorithm to provide a wrong classification for the inspected code and try to counter such an option with techniques such as continuously updating the classifier's training set, the same way signatures DB are being updated in current anti-virus programs, or applying the input transformation discussed. However, as we have shown, even such transformations are susceptible to camouflage algorithms designed against them. Another alternative is to use multiple classifiers simultaneously in order to reduce the camouflage risk. Finally, one can train a classifier to detect the camouflage instead-of maliciousness, e.g., by finding a sequence of system calls which is unreasonable.

Our future work in this area would examine those mitigation possibilities. We would also investigate the effectiveness of our camouflage algorithm on other machine-learning classifiers (e.g. SVM, boosted trees, etc.) and find other algorithms to cope with such classifiers. We would also like to research camouflage algorithms for classifiers with other input types besides system call sequences (e.g. call flow graphs), for classifiers with more than a single input type and for ensemble learning.

References

1. Baldi P., Brunak S., Chauvin Y., Andersen CA., Nielsen H.: Assessing the accuracy of prediction algorithms for classification: an overview. In: Bioinformatics, Volume 16 Issue 5, pp. 412-24 (2000)
2. Barreno M., Nelson B., Sears R., Joseph A., Tygar J.: Can machine learning be secure? In: ASIACCS'06, pp. 16–25 (2006)
3. Barreno M., Nelson B., Joseph A., Tygar J.: The security of machine learning. Machine Learning, Vol 81, pp. 121–148 (2010)
4. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: Proceedings of the 15th Ann. Conf. European Inst. for Computer Antivirus Research, pp. 180–192 (2006)
5. Biggio, B., Pillai, I., Rota Bul'o, S., Ariu, D., Pelillo, M., Roli, F.: Is data clustering in adversarial settings secure? In: Proceedings of the 6th ACM Workshop on Artificial Intelligence and Security (2013)
6. Biggio B., Corona I., Maiorca D., Nelson B., Srndic N., Laskov P., Giacinto G., Roli F.: Evasion attacks against machine learning at test time. In: European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD), Part III, volume 8190 of Lecture Notes in Computer Science, pp. 387–402 (2013)
7. Biggio, B., Rieck, K., Ariu, D., Wressnegger, C., Corona, I., Giacinto, G., Rol., F.: Poisoning behavioral malware clustering. In: Proceedings of the 7th ACM Workshop on Artificial Intelligence and Security (2014)
8. Biggio B., Corona I., Nelson B., Rubinstein B., Maiorca D., Fumera G., Giacinto G., Roli F.: Security evaluation of support vector machines in adversarial environments. In: Support Vector Machines Applications, pp. 105–153 (2014)
9. Bellard F.: QEMU, A Fast and Portable Dynamic Translator. Proc. Usenix 2005 Ann. Technical Conf. (Usenix '05), Usenix Assoc., pp. 41–46 (2005)
10. Breiman, L: Random Forests. In: Machine Learning, 45(1), pp. 5-32 (2001)
11. Caruana, R., Niculescu-Mizil, A.: An Empirical Comparison of Supervised Learning Algorithms. ICML '06, Proceedings of the 23rd international conference on Machine learning, pp. 161-168.
12. Crawford, S. L.: Extensions to the CART algorithm. In: International journal of man-machine studies, vol. 31, pp. 197-217 (1989)
13. Dalvi N., Domingos P., Mausam, Sanghai S., Verma D.: Adversarial classification. In KDD '04: Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 99–108 (2004)
14. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. In: ACM Computing Surveys, Vol. 44, No. 2, Article 6, pp. 1-42 (2012)
15. Ferrie P.: Attacks on Virtual Machine Emulators. Symantec Advanced Threat Research.
16. Ferrie P.: Attacks on More Virtual Machine Emulators. Symantec Advanced Threat Research.
17. Firdausi, I., Lim, C., Erwin, A.: Analysis of Machine Learning Techniques Used in Behavior Based Malware Detection. In: Proceedings of 2nd International Conference on Advances in Computing, Control and Telecommunication Technologies, pp. 201-203 (2010)
18. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longsta, T.A.: A Sense of Self for Unix Processes. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 120-128, IEEE Press, USA (1996)

19. Forrest, S., Hofmeyr, S., Somayaji, A.: The evolution of system-call monitoring. In: Proceedings of the Annual Computer Security Applications Conference, pp. 418–430 (2008)
20. Gambs, S., Gmati, A., Hurfin, M.: Reconstruction attack through classifier analysis. In: Proceedings of the 26th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy, pp. 274-281 (2012)
21. Gao, D., Reiter, M., Song, D.: On gray-box program tracking for anomaly detection. In: USENIX Security Symposium, pp. 103–118 (2004)
22. Guyon, I., Elisseeff, A.: An introduction to Variable and Feature Selection. In: Journal of Machine Learning Research 3, pp. 1157-1182 (2003)
23. Hamlen, K.W., Mohan, V., Masud, M.M., Khan L., Thuraisingham B.: Exploiting an Antivirus Interface. In: Computer Standards & Interfaces, Volume 31 Issue 6, pp. 1182-1189 (2009)
24. Huang L., Joseph A., Nelson B., Rubinstein B., Tygar J.: Adversarial machine learning. In: 4th ACM Workshop on Artificial Intelligence and Security, pp. 43–57 (2011)
25. Hunt, G.C., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: Proceedings of the 3rd Usenix Windows NT Symp., pp. 135–143 (1999)
26. King, S.T., Chen, P.M., Wang, Y.M., Verbowski, C., Wang, H.J., Lorch, J.R.: SubVirt: Implementing Malware with Virtual Machines. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 314– 327 (2006)
27. Kolter J.Z., Maloof M.A.: Learning to detect and classify malicious executables in the wild. In: Journal of Machine Learning Research, 7(Dec), pp. 2721 – 2744 (2006)
28. Kolter, J.Z., Maloof, M.A.: Learning to detect malicious executables in the wild. In: Proceedings of the 10th International Conference on Knowledge Discovery and Data Mining, pp. 470–478 (2004)
29. Lakhotia, A., Preda, M.D., Giacobazzi, R.: Fast Location of Similar Code Fragments using Semantic 'Juice'. In: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, (2013)
30. Lowd D., Meek C.: Adversarial learning. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp. 641–647 (2005)
31. Ming, J., Xin, Z., Lan, P., Wu, D., Liu, P., Mao, B.: Replacement Attacks: Automatically Impeding Behavior-based Malware Specifications. In: Proceedings of the 13th International Conference on Applied Cryptography and Network Security (2015)
32. Moser, A., Kruegel, C., Kirda, E.: Limits of Static Analysis for Malware Detection. In: 23rd Annual Computer Security Applications Conference, pp. 421-430 (2007)
33. Moskovitch, R., Gus, I., Pluderman, S., Stopel, D., Fermat, Y., Shahar, Y., Elovici, Y.: Host Based Intrusion Detection Using Machine Learning. In: Proceedings of Intelligence and Security Informatics, pp. 107-114 (2007)
34. Moskovitch R., Nissim N., Stopel D., Feher C., Englert R., Elovici Y.: Improving the Detection of Unknown Computer Worms Activity Using Active Learning. In: Proceedings of the 30th annual German conference on Advances in Artificial Intelligence, pp. 489 – 493 (2007)
35. Navarro, G.: A guided tour to approximate string matching. In: ACM Computing Surveys, vol. 33, no. 1, pp. 31-88 (2001)
36. Ouellette, J., Pfeffer, A., Lakhotia, A.: Countering malware evolution using cloud-based learning. In: Malicious and Unwanted Software: 'The Americas' (MALWARE), 2013 8th International Conference, pp. 85-94 (2013)
37. Parampalli, C., Sekar, R., Johnson, R.: A Practical Mimicry Attack Against Powerful System-Call Monitors. In: Proceedings of the ACM Symposium on Information, Computer and Communications Security, pp. 156-167 (2008)

38. Powers D.: Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. In: Journal of Machine Learning Technologies 2 (1), pp. 37–63 (2011)
39. Quinlan, J.R.: Discovering rules from large collections of examples: A case study. In: Expert Systems in the Microelectronic Age, Michie, D. (Ed.), pp. 168-201 (1979)
40. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting System Emulators. In: Proceedings of Information Security, 10th International Conference, pp. 1-18 (2007)
41. Rosenberg I., Gudes, E.: Attacking and Defending Dynamic Analysis System-Calls Based IDS. In: Information Security Theory and Practice (WISTP), 10th International Conference, pp. 103-119 (2016)
42. Rosenberg I., Gudes E.: Evading System-Calls Based Intrusion Detection Systems. In: Network and System Security (NSS) 10th International Conference, pp. 200-216 (2016)
43. Rozenberg, B., Gudes, E., Elovici, Y., Fledel, Y.: Method for Detecting Unknown Malicious Executables. In: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection, pp. 376-377 (2009)
44. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. In: Information Sciences 227, pp. 63–81 (2013)
45. Shabtai, A., Menahem, E., Elovici, Y.: F-Sign: Automatic, Function-Based Signature Generation for Malware. In: IEEE Trans. Systems, Man, and Cybernetics—Part C: Applications and Reviews, vol. 41, no. 4, pp. 494-508 (2011)
46. Schultz M., Eskin E., Zadok E., Stolfo S.: Data mining methods for detection of new malicious executables. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 38–49 (2001)
47. Singhal, P., Raul, N.: Malware Detection Module Using Machine Learning Algorithms to Assist in Centralized Security in Enterprise Networks. In: International Journal of Network Security & Its Applications, Vol.4, No.1, pp. 661-67 (2012)
48. Somayaji, A., Forrest, S.: Automated Response Using System-Call Delays. In: Proceedings of the 9th USENIX Security Symposium, pp. 185-198 (2000)
49. Srndic N., Laskov P.: Practical evasion of a learning- based classifier - A case study. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, pp. 197–211 (2014)
50. Sufatrio, Yap, R. H. C.: Improving Host-Based IDS with Argument Abstraction to Prevent Mimicry Attacks. In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection, pp. 146–164 (2005)
51. Tan, K., Killourhy, K., Maxion, R.: Understanding an Anomaly-Based Intrusion Detection System Using Common Exploits. In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection(2002)
52. Tan K., McHugh J., Killourhy K.: Hiding intrusions: from the abnormal to the normal and beyond. In: Proceedings of Information Hiding: 5th International Workshop, pp. 1–17 (2003)
53. Tandon, G., Chan, P.: Learning Rules from System Call Arguments and Sequences for Anomaly Detection. In: Proceedings of ICDM Workshop on Data Mining for Computer Security, pp. 20-29, IEEE Press, USA (2003)
54. Tandon, G., Chan. P.: On the Learning of System Call Attributes for Host-Based Anomaly Detection. In: International Journal on Artificial Intelligence Tools, 15(6), pp. 875–892 (2006)
55. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: Proceedings of the IEEE Symposium on Security and Privacy (2001)

56. Wagner, D., Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems. In: Proceedings of the 9th ACM conference on Computer and Communications Security, pp. 255-264 (2002)
57. Warrender C., Forrest S., Pearlmutter B.: Detecting Intrusions Using System Calls: Alternative data models. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp. 133–145 (1999)
58. Willems, C., Holz, T., Freiling, F.: Toward Automated Dynamic Malware Analysis Using CWSandbox. IEEE: Security & Privacy, Volume 5 , Issue: 2, pp. 32 – 39 (2007)
59. Xiao H., Biggio B., Nelson B., H. Xiao H., Eckert C., Roli F.: Support vector machines under adversarial label contamination. In: Neurocomputing, Special Issue on Advances in Learning with Label Noise, Vol. 160, pp. 53-62 (2014)
60. Yang, Y., Fanglu, G., Susanta, N., Lap-chung, L., Tzi-cker, C.: A Feather-weight Virtual Machine for Windows Applications. In: Proceedings of the 2nd International Conference on Virtual Execution Environments, pp. 24-34 (2006)
61. Yates, F.: Contingency table involving small numbers and the χ 2 test. In: Journal of the Royal Statistical Society, vol. 1-2, pp. 217-235 (1934)
62. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: Proceedings of the 14th ACM Conference of Computer and Communication Security, pp. 116-127 (2007)
63. Zheng Z., Wu X., Srihari R.: Feature Selection for Text Categorization on Imbalanced Data. SIGKDD Explorations, pp. 80-89 (2002)
64. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. In: SIAM Journal of Computing, 18, pp. 1245–1262 (1989)

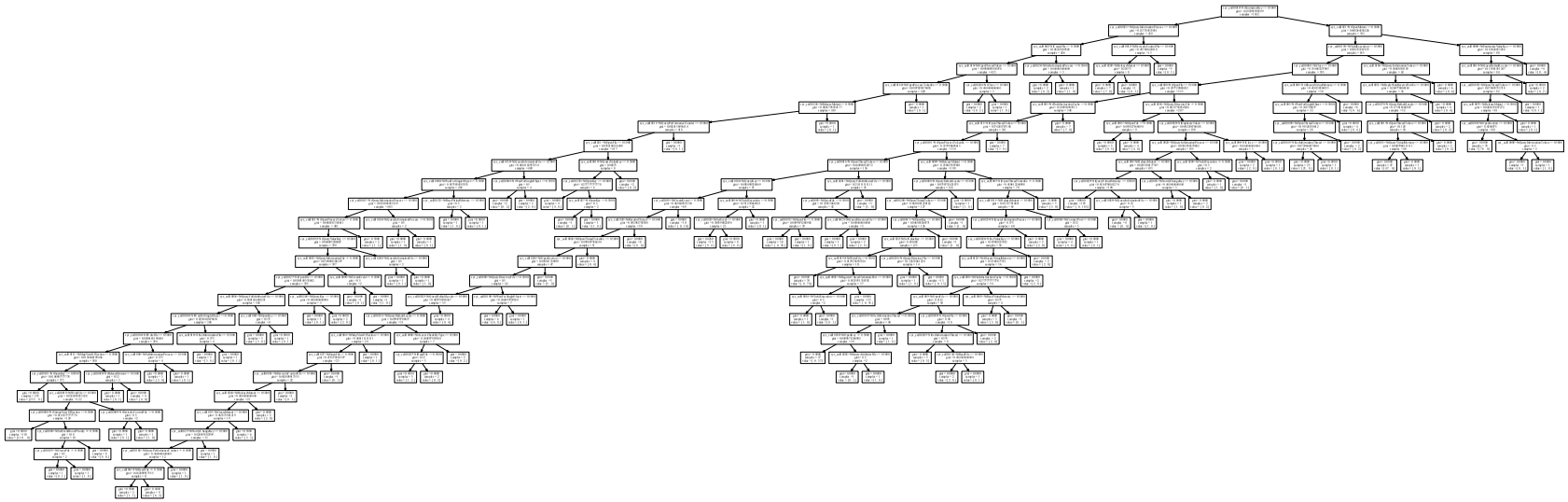
A Appendix A: Sample decision trees used by the IDS classifier

Attached are two decision trees created from the training set, one with and one without input transformation.

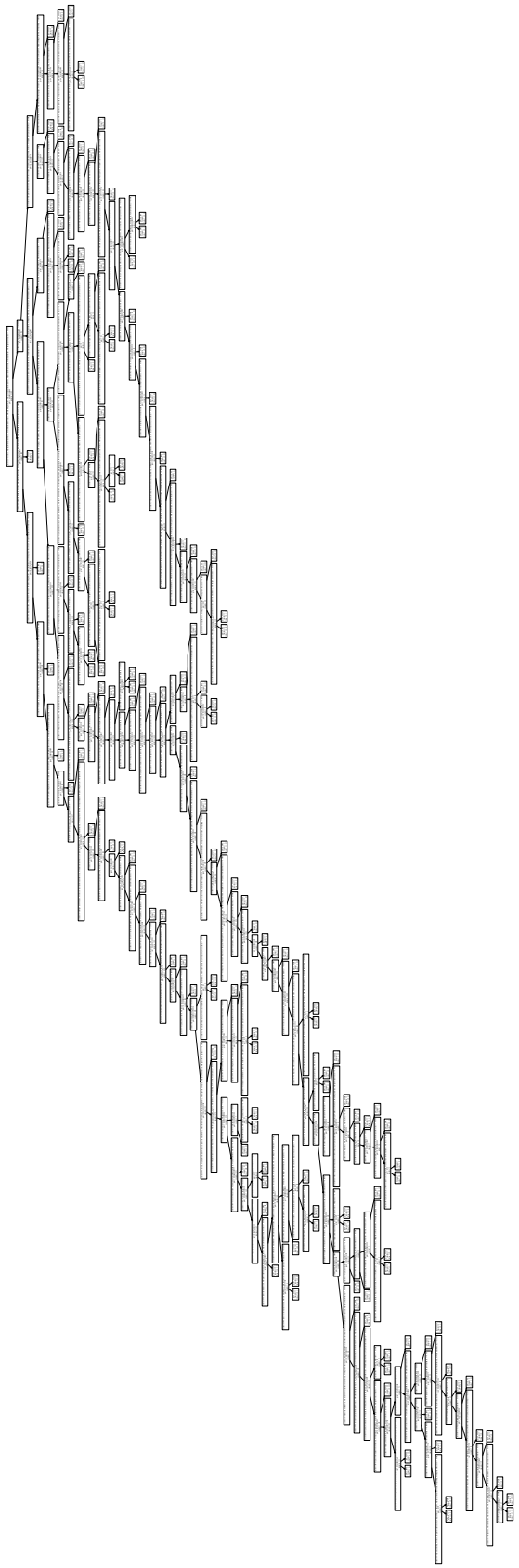
If the decision tree condition specified in a node is, e.g.: sys_call10018=NtEnumerateKey, this means the right child of this node in the decision path is chosen if *sys_call[10018]=NtEnumerateKey,* and the left child is chosen if not.

A leaf node has a value vector of *[benign_training_set_samples, malicious_training_set_samples]*. Thus, an inspected code would be classified as malicious if |*malicious_training_set_samples*| ¿ |*benign_training_set_samples*| at that node, or as benign otherwise.

A.1 Decision Tree without input transformation



A.2 Decision Tree with section-based input transformation, order-preserving, with duplicates removal



B Appendix B: Computing Levenshtein Distance

B.1 Recursive

A straightforward, but inefficient, recursive pseudo-code implementation of a LevenshteinDistance function that takes two strings, s and t, together with their lengths, and returns the Levenshtein distance between them:

```
// len_s and len_t are the number of characters in string s and t respectively
int LevenshteinDistance(string s, int len_s, string t, int len_t)
{
  /* base case: empty strings */
  if (len_s == 0) return len_t;
  if (len_t == 0) return len_s;
  /* test if last characters of the strings match */
  if (s[len_s-1] == t[len_t-1])
    cost = 0;
  else
    cost = 1;
  /* return minimum of delete char from s, delete char from t, and delete char
from both */
  return minimum(LevenshteinDistance(s, len_s - 1, t, len_t ) + 1,
    LevenshteinDistance(s, len_s , t, len_t - 1) + 1,
    LevenshteinDistance(s, len_s - 1, t, len_t - 1) + cost);
}
```

B.2 Iterative with Full Matrix

Computing the Levenshtein distance is based on the observation that if we reserve a matrix to hold the Levenshtein distances between all prefixes of the first string and all prefixes of the second, then we can compute the values in the matrix in a dynamic programming fashion, and thus find the distance between the two full strings as the last value computed:

```
function LevenshteinDistance(char s[1..m], char t[1..n]):
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t;
  // note that d has (m+1)*(n+1) values
  declare int d[0..m, 0..n]
  set each element in d to zero
  // source prefixes can be transformed into empty string by
  // dropping all characters
  for i from 1 to m:
    d[i, 0] := i
  // target prefixes can be reached from empty source prefix
  // by inserting every character
  for j from 1 to n:
```

```
    d[0, j] := j
  for j from 1 to n:
    for i from 1 to m:
      if s[i] = t[j]:
        d[i, j] := d[i-1, j-1] // no operation required
      else:
        d[i, j] := minimum(d[i-1, j] + 1, // a deletion
          d[i, j-1] + 1, // an insertion
          d[i-1, j-1] + 1) // a substitution
  return d[m, n]
```

Example 9. The Levenshtein distance between 'Sunday' and 'Saturday':

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

B.3 Iterative with Two Matrix Rows

It turns out that only two rows of the table are needed for the construction if one does not want to reconstruct the edited input strings (the previous row and the current row being calculated).

The Levenshtein distance may be calculated iteratively using the following algorithm:

```
int LevenshteinDistance(string s, string t)
{
  // degenerate cases
  if (s == t) return 0;
  if (s.Length == 0) return t.Length;
  if (t.Length == 0) return s.Length;
  // create two work vectors of integer distances
  int[] v0 = new int[t.Length + 1];
  int[] v1 = new int[t.Length + 1];
  // initialize v0 (the previous row of distances)
  // this row is A[0][i]: edit distance for an empty s
  // the distance is just the number of characters to delete from t
  for (int i = 0; i ¡ v0.Length; i++)
```

```
        v0[i] = i;
    for (int i = 0; i ¡ s.Length; i++)
    {
        // calculate v1 (current row distances) from the previous row v0
        // first element of v1 is A[i+1][0]
        // edit distance is delete (i+1) chars from s to match empty t
        v1[0] = i + 1;
        // use formula to fill in the rest of the row
        for (int j = 0; j ¡ t.Length; j++)
        {
            var cost = (s[i] == t[j]) ? 0 : 1;
            v1[j + 1] = Minimum(v1[j] + 1, v0[j + 1] + 1, v0[j] + cost);
        }
        // copy v1 (current row) to v0 (previous row) for next iteration
        for (int j = 0; j ¡ v0.Length; j++)
            v0[j] = v1[j];
    }
    return v1[t.Length];
}
```

C Appendix C: Levenshtein Distance: Inferring the edit operations from the matrix

Given the matrix produced by the Levenshtein algorithm (see Appendix B), how can we find the precise sequence of string operations: inserts, deletes and substitution [of a single letter], necessary to convert the 's string' into the 't string'?

By "decoding" the Levenshtein matrix, one can enumerate ALL such optimal sequences. The general idea is that the optimal solutions all follow a "path", from top left corner to bottom right corner (or in the other direction), whereby the matrix cell values on this path either remain the same or increase by one (or decrease by one in the reverse direction), starting at 0 and ending at the optimal number of operations for the strings in question. The number increases when an operation is necessary, it stays the same when the letter at corresponding positions in the strings are the same.

This path finding algorithm should start at the lower right corner and work its way backward. The reason for this approach is that we know for a fact that to be an optimal solution it must end in this corner, and to end in this corner, it must have come from one of the 3 cells either immediately to its left, immediately above it or immediately diagonally. By selecting a cell among these three cells, one which satisfies our "same value or decreasing by one" requirement, we effectively pick a cell on one of the optimal paths. By repeating the operation till we get on upper left corner (or indeed until we reach a cell with a 0 value), we effectively backtrack our way on an optimal path.

The conventions are:

- An horizontal move is an INSERTION of a letter from the 't string'
- A vertical move is a DELETION of a letter from the 's string'
- A diagonal move is either:
  - A no-operation (both letters at respective positions are the same); the number doesn't change
  - A SUBSTITUTION (letters at respective positions are distinct); the number increase by one

One possible approach to select one path among several possible optimal paths is loosely described below:

Starting at the bottom-rightmost cell, and working our way backward toward the top left.

For each "backward" step, consider the 3 cells directly adjacent to the current cell (in the left, top or left+top directions)

if the value in the diagonal cell (going up+left) is smaller or equal to the values found in the other two cells AND if this is same or 1 minus the value of the current cell

then "take the diagonal cell"

if the value of the diagonal cell is one less than the current cell:

Add a SUBSTITUTION operation (from the letters corresponding to the _current_ cell)

otherwise: do not add an operation this was a no-operation.

elseif the value in the cell to the right is smaller of equal to the value of the of the cell above current cell AND if this value is same or 1 minus the value of the current cell

then “take the cell to right”, and add an INSERTION of the letter corresponding to the cell

else

take the cell above, add

Add a DELETION operation of the letter in ’s string’

Example 10. Following this informal pseudo-code, we get the following:

		r	e	p	u	b	l	i	c	a	n
	0	1	2	3	4	5	6	7	8	9	10
d	1	1	2	3	4	5	6	7	8	9	10
e	2	2	1	2	3	4	5	6	7	8	9
m	3	3	2	2	3	4	5	6	7	8	9
o	4	4	3	3	3	4	5	6	7	8	9
c	5	5	4	4	4	4	5	6	6	7	8
r	6	5	5	5	5	5	5	6	7	7	8
a	7	6	6	6	6	6	6	6	7	7	8
t	8	7	7	7	7	7	7	7	7	8	8

Start on the “n”, “t” cell at bottom right.

Pick the [diagonal] “a”, “a” cell as next destination since it is less than the other two (and satisfies the same or -1 condition).

Note that the new cell is one less than current cell

therefore the step 8 is substitute “t” with “n”: democra N

Continue with “a”, “a” cell,

Pick the [diagonal] “c”, “r” cell as next destination...

Note that the new cell is same value as current cell ==¿ no operation needed.

Continue with “c”, “r” cell,

Pick the [diagonal] “i”, “c” cell as next destination...

Note that the new cell is one less than current cell

therefore the step 7 is substitute “r” with “c”: democ C an

Continue with “i”, “c” cell,

Pick the [diagonal] “l”, “o” cell as next destination...

Note that the new cell is one less than current cell

therefore the step 6 is substitute “c” with “i”: demo I can

Continue with “l”, “o” cell,

Pick the [diagonal] “b”, “m” cell as next destination...

Note that the new cell is one less than current cell

therefore the step 5 is substitute “o” with “l”: dem L ican

Continue with “b”, “m” cell,
Pick the [diagonal]”u”, “e” cell as next destination...
Note that the new cell is one less than current cell
therefore the step 4 is substitute “m” with “b”: de B lican

Continue with “u”, “e” cell,
Note the “diagonal” cell doesn’t qualify, because the “left” cell is less than it. Pick the [left] “p”, “e” cell as next destination...
therefore the step 3 is instert “u” after “e”: de U blican

Continue with “p”, “e” cell,
again the “diagonal” cell doesn’t qualify Pick the [left] “e”, “e” cell as next destination...
therefore the step 2 is instert “p” after “e”: de P ublican

Continue with “e”, “e” cell,
Pick the [diagonal] “r”, “d” cell as next destination...
Note that the new cell is same value as current cell ==¿ no operation needed.

Continue with “r”, “d” cell,
Pick the [diagonal] “start” cell as next destination...
Note that the new cell is one less than current cell
therefore the step 1 is substitute “d” with “r”: R epublican

You’ve arrived at a cell which value is 0 : The operations were found.

D Appendix D: Various Machine Learning Classifiers - Mathematical and Technical Background

D.1 Random-Forest Classifier

Decision trees that are grown very deep tend to learn highly irregular patterns: they overfit their training sets, because they have low bias, but very high variance.

Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set. When splitting a node during the construction of the tree, the split that is chosen is no longer the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance also decreases, usually more than compensating for the increase in bias, hence yielding an overall better model.

D.1.1 Tree bagging The training algorithm for random forests applies the general technique of bootstrap aggregating, or bagging, to tree learners. Given a training set X = x1, . . . , xn with responses Y = y1, . . . , yn, bagging repeatedly selects a random sample with replacement of the training set and fits trees to these samples:

1. For b = 1, . . . , B:
   a) Sample, with replacement, n training examples from X, Y; call these Xb, Yb.
   b) Train a decision or regression tree fb on Xb, Yb.

After training, predictions for unseen samples x' is made by taking the majority vote in the case of decision trees.

As already mentioned, this bootstrapping procedure leads to better model performance because it decreases the variance of the model, without increasing the bias, or with a minimal increase in total. This means that while the predictions of a single tree are highly sensitive to noise in its training set, the average of many trees is not, as long as the trees are not correlated. Simply training many trees on a single training set would give strongly correlated trees (or even the same tree many times, if the training algorithm is deterministic); bootstrap sampling is a way of de-correlating the trees by training them with different training sets.

The number of samples/trees, B, is a free parameter.

D.1.2 Feature Bagging The above procedure describes the original bagging algorithm for trees. Random forests differ in only one way from this general scheme: they use a modified tree learning algorithm that selects, at each candidate split in the learning process, a random subset of the features. This process is sometimes called "feature bagging". The reason for doing this is the correlation of the trees in an ordinary bootstrap sample: if one or a few features are very strong predictors for the response variable (target output), these features will be selected in many of the decision trees, causing them to become correlated.

D.1.3 Implementation The IDS classifier was implemented using scikit-learn's sklearn.ensemble.RandomForestClassifier class[17]. The default value is B=10 decision trees, using a default metric of Gini impurity.

D.2 K-Nearest Neighbors (k-NN) Classifier

The input of the k-Nearest Neighbors algorithm consists of the k closest training examples in the feature space. The output is a class membership. An object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. k-NN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification.

A shortcoming of the k-NN algorithm is that it is sensitive to the local structure of the data.

The training examples are vectors in a multidimensional feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples.

In the classification phase, k is a user-defined constant, and an unlabeled vector (a query or test point) is classified by assigning the label which is most frequent among the k training samples nearest to that query point.

The used distance metric is Euclidean distance.

D.2.1 Implementation The IDS classifier was implemented using scikit-learn's sklearn.neighbors.KNeighborsClassifier class[18]. The default values are: k=5, distance metric=standard Euclidean metric.

D.3 Naïve Bayes (NB) Classifier

The classifier is based upon a naïve assumption, by which it is named: Even if features depend upon the existence of others, they independently contribute to the classification.

A pro of this classifier: It requires a small amount of training data to estimate the parameters (means and variances of the variables) necessary for classification

A con of this classifier: Bayes classification is outperformed by approaches such as random forest and SVM (as mentioned in [11]).

D.3.1 The NB Probability Model naive Bayes is a conditional probability model: given a problem instance to be classified, represented by a vector $\mathbf{x} = (x_1, \ldots, x_n)$ representing some n features (dependent variables), it assigns to this instance probabilities

$p(C_k|x_1, \ldots, x_n)$

for each of k possible outcomes or classes.

[17] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[18] http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Using Bayes' theorem, under the above independence assumptions, the conditional distribution over the class variable C is:

$p(C_k|x_1,\ldots,x_n) = \frac{1}{Z} p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$

where the evidence $Z = p(\mathbf{x})$ is a scaling factor dependent only on $x_1,\ldots,x_n$, that is, a constant if the values of the feature variables are known.

D.3.2 The NB Classifier This classifier pick the hypothesis that is most probable; this is known as the maximum a posteriori or MAP decision rule.

The corresponding classifier, a Bayes classifier, is the function that assigns a class label $\hat{y} = C_k$ for some k as follows:

$$\hat{y} = \underset{k \in \{1,\ldots,K\}}{argmax}\ p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

D.3.3 Gaussian Naïve Bayes A typical assumption is that the values associated with each class are distributed according to a Gaussian distribution. For example, suppose the training data contain a continuous attribute, x. We first segment the data by the class, and then compute the mean and variance of x in each class. Let μ_c be the mean of the values in x associated with class c, and let σ_c^2 be the variance of the values in x associated with class c. Then, the probability distribution of some value given a class, $p(x = v|c)$, can be computed by plugging v into the equation for a Normal distribution parameterized by μ_c and σ_c^2. That is,

$$p(x = v|c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{-\frac{(v-\mu_c)^2}{2\sigma_c^2}}$$

D.3.4 Bernoulli Naïve Bayes In the multivariate Bernoulli event model, features are independent booleans (binary variables) describing inputs. If x_i is a boolean expressing the occurrence or absence of the i'th term from the vocabulary, then the likelihood of a document given a class C_k is given by:

$p(\mathbf{x}|C_k) = \prod_{i=1}^{n} p_{ki}^{x_i} (1 - p_{ki})^{(1-x_i)}$

where p_{ki} is the probability of class C_k generating the term x_i.

D.3.5 Implementation The IDS Gaussian NB classifier was implemented using scikit-learn's sklearn.naive_bayes.GaussianNB class[19].

The IDS Bernoulli NB classifier was implemented using scikit-learn's sklearn.naive_bayes.BernoulliNB class[20].

D.4 AdaBoost Classifier

AdaBoost, short for "Adaptive Boosting", can be used in conjunction with many other types of learning algorithms to improve their performance. The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the boosted classifier. AdaBoost is adaptive in the sense

[19] http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

[20] http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html

that subsequent weak learners are tweaked in favor of those instances misclassified by previous classifiers. The core principle of AdaBoost is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights $w_1, w_2, ..., w_N$ to each of the training samples. Initially, those weights are all set to $w_i = 1/N$, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence. AdaBoost (with decision trees as the weak learners) is often referred to as the best out-of-the-box classifier. Unlike neural networks and SVMs, the AdaBoost training process selects only those features known to improve the predictive power of the model, reducing dimensionality and potentially improving execution time as irrelevant features do not need to be computed.

D.4.1 Training AdaBoost refers to a particular method of training a boosted classifier. A boost classifier is a classifier in the form

$F_T(x) = \sum_{t=1}^{T} f_t(x)$

where each f_t is a weak learner that takes an object x as input and returns a real valued result indicating the class of the object. The sign of the weak learner output identifies the predicted object class and the absolute value gives the confidence in that classification. Similarly, the T-layer classifier will be positive if the sample is believed to be in the positive class and negative otherwise.

Each weak learner produces an output, hypothesis $h(x_i)$, for each sample in the training set. At each iteration t, a weak learner is selected and assigned a coefficient α_t such that the sum training error E_t of the resulting t-stage boost classifier is minimized.

$E_t = \sum_i E[F_{t-1}(x_i) + \alpha_t h(x_i)]$

Here $F_{t-1}(x)$ is the boosted classifier that has been built up to the previous stage of training, $E(F)$ is some error function and $f_t(x) = \alpha_t h(x)$ is the weak learner that is being considered for addition to the final classifier.

D.4.2 Weighting At each iteration of the training process, a weight is assigned to each sample in the training set equal to the current error $E(F_{t-1}(x_i))$ on that sample. These weights can be used to inform the training of the weak learner, for instance, decision trees can be grown that favor splitting sets of samples with high weights.

D.4.3 Discrete AdaBoost The discrete variant, used in our IDS, is as follows:

With:

– Samples $x_1 \ldots x_n$
– Desired outputs $y_1 \ldots y_n, y \in \{-1, 1\}$
– Initial weights $w_{1,1} \ldots w_{n,1}$ set to $\frac{1}{n}$
– Error function $E(f(x), y, i) = e^{-y_i f(x_i)}$
– Weak learners $h\colon x \to [-1, 1]$

For t in $1 \ldots T$:

1. Choose $f_t(x)$:
   a) Find weak learner $h_t(x)$ that minimizes ε_t, the weighted sum error for misclassified points $\varepsilon_t = \sum_i w_{i,t} E(h_t(x), y, i)$
   b) Choose $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\varepsilon_t}{\varepsilon_t}\right)$
2. Add to ensemble:
   a) $F_t(x) = F_{t-1}(x) + \alpha_t h_t(x)$
3. Update weights:
   a) $w_{i,t+1} = w_{i,t} e^{-y_i \alpha_t h_t(x_i)}$ for all i
   b) Renormalize $w_{i,t+1}$ such that $\sum_i w_{i,t+1} = 1$

D.4.4 Implementation The IDS classifier was implemented using scikit-learn's sklearn.ensemble.AdaBoostClassifier class[21]. The default value is a maximum of 50 estimators for boosting (or less if a perfect fit is achieved before).

D.5 Support Vector Machine (SVM) Classifier

A data point is viewed as a $p$dimensional vector (a list of p numbers), and we want to know whether we can separate such points with a $(p-1)$dimensional hyperplane. This is called a linear classifier. There are many hyperplanes that might classify the data. One reasonable choice as the best hyperplane is the one that represents the largest separation, or margin, between the two classes. So we choose the hyperplane so that the distance from it to the nearest data point on each side is maximized. If such a hyperplane exists, it is known as the maximum-margin hyperplane and the linear classifier it defines is known as a maximum margin classifier; or equivalently, the perception of optimal stability.

An example of such separation in Figure 5. H3 does not separate the classes. H1 does, but only with a small margin. H2 separates them with the maximum margin.

[21] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

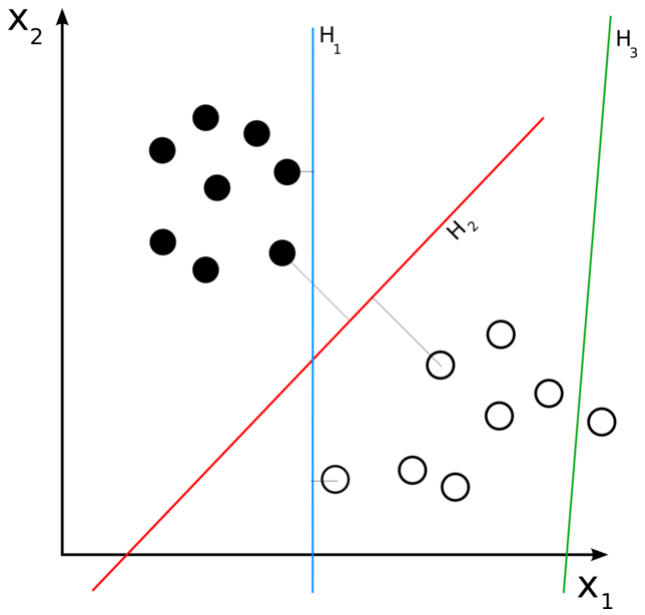


Fig. 5. Example of Separating Hyperplanes

D.5.1 Linear SVM Given some training data $\mathscr{D}$, a set of n points of the form

$\mathscr{D} = \{(\mathbf{x}_i, y_i) \mid \mathbf{x}_i \in R^p, y_i \in \{-1, 1\}\}_{i=1}^n$

where the y_i is either 1 or −1, indicating the class to which the point $\mathbf{x}_i$ belongs. Each $\mathbf{x}_i$ is a $p$dimensional real vector. We want to find the maximum-margin hyperplane that divides the points having $y_i = 1$ from those having $y_i = -1$. Any hyperplane can be written as the set of points $\mathbf{x}$ satisfying:

$\mathbf{w} \cdot \mathbf{x} - b = 0$

where $\cdot$ denotes the dot product and $\mathbf{w}$ the (not necessarily normalized) normal vector to the hyperplane. The parameter $\frac{b}{\|\mathbf{w}\|}$ determines the offset of the hyperplane from the origin along the normal vector $\mathbf{w}$.

If the training data are linearly separable, we can select two hyperplanes in a way that they separate the data and there are no points between them, and then try to maximize their distance. The region bounded by them is called "the margin". These hyperplanes can be described by the equations

$\mathbf{w} \cdot \mathbf{x} - b = 1$

and

$\mathbf{w} \cdot \mathbf{x} - b = -1$

By using geometry, we find the distance between these two hyperplanes is $\frac{2}{\|\mathbf{w}\|}$, so we want to minimize $\|\mathbf{w}\|$. As we also have to prevent data points from falling into the margin, we add the following constraint: for each i either

$\mathbf{w} \cdot \mathbf{x}_i - b \geq 1$ for $\mathbf{x}_i$of the first class

or

$\mathbf{w} \cdot \mathbf{x}_i - b \leq -1$ for $\mathbf{x}_i$of the second.

This can be rewritten as:

$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1, \text{ for all } 1 \leq i \leq n$

We can put this together to get the optimization problem:

Minimize (in $\mathbf{w}, b$)

$\|\mathbf{w}\|$

subject to (for any $i = 1, \ldots, n$)

$y_i(\mathbf{w} \cdot \mathbf{x_i} - b) \geq 1$

In Figure 6 we see a maximum-margin hyperplane and margins for an SVM trained with samples from two classes. Samples on the margin are called the support vectors.

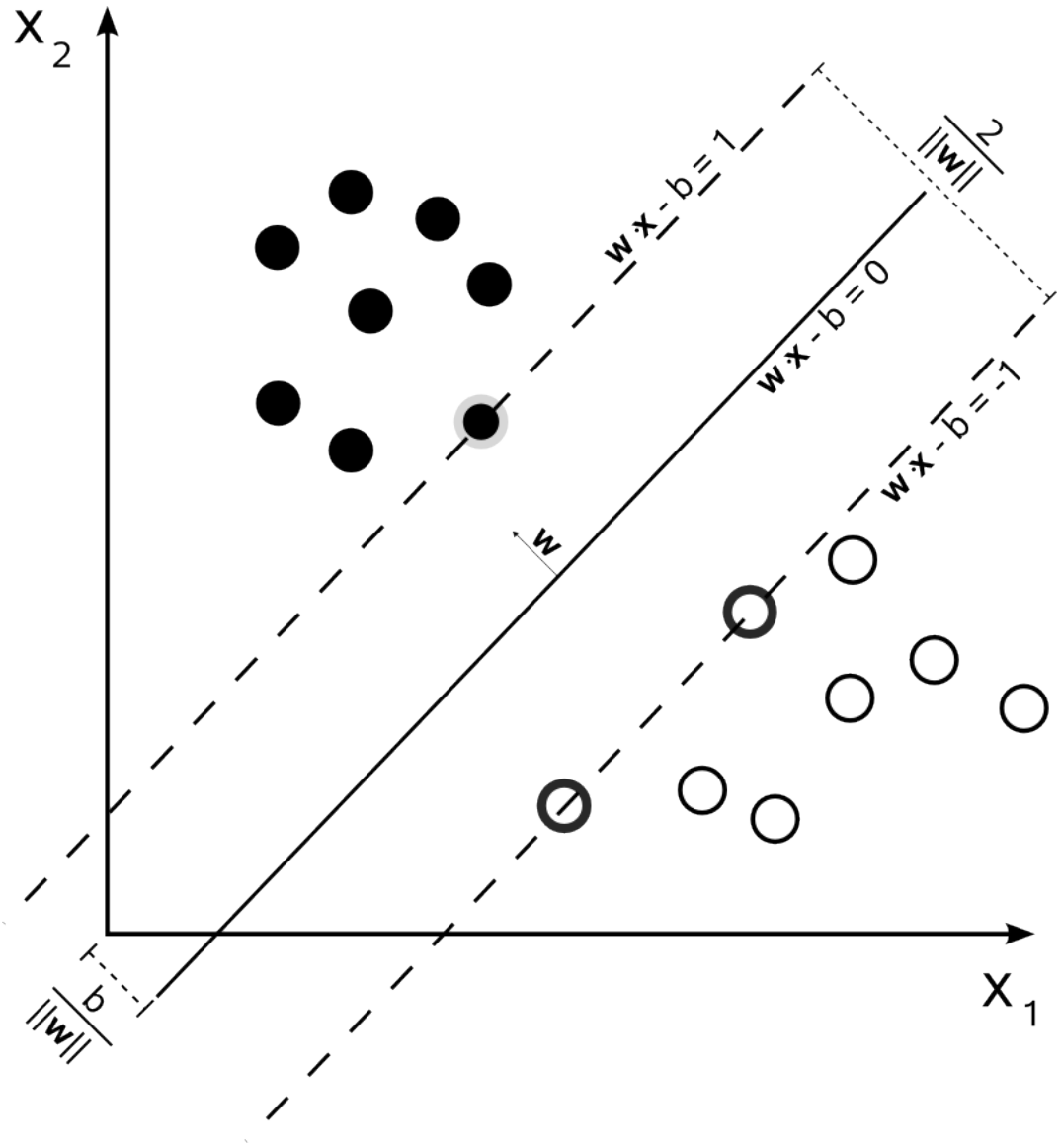


Fig. 6. Example of Support Vectors

D.5.2 Nonlinear classification A way to create nonlinear classifiers is by applying the kernel trick to maximum-margin hyperplanes. The resulting algorithm is formally similar, except that every dot product is replaced by a nonlinear kernel function. This allows the algorithm to fit the maximum-margin hyperplane in a transformed feature space. The transformation may be nonlinear and the trans-

formed space high dimensional; thus though the classifier is a hyperplane in the high-dimensional feature space, it may be nonlinear in the original input space.

An example of such a kernel function is a Gaussian Radial Basis Function (RBF):

$k(\mathbf{x_i}, \mathbf{x_j}) = \exp(-\gamma \|\mathbf{x_i} - \mathbf{x_j}\|^2)$, for $\gamma > 0$. Sometimes parametrized using $\gamma = 1/2\sigma^2$

If the kernel used is a Gaussian radial basis function, the corresponding feature space is a Hilbert space of infinite dimensions.

he kernel is related to the transform $\varphi(\mathbf{x_i})$ by the equation $k(\mathbf{x_i}, \mathbf{x_j}) = \varphi(\mathbf{x_i}) \cdot \varphi(\mathbf{x_j})$. The value w is also in the transformed space, with $\mathbf{w} = \sum_i \alpha_i y_i \varphi(\mathbf{x}_i)$. Dot products with w for classification can again be computed by the kernel trick, i.e. $\mathbf{w} \cdot \varphi(\mathbf{x}) = \sum_i \alpha_i y_i k(\mathbf{x}_i, \mathbf{x})$

An example for the kernel trick is shown in Figure 7.

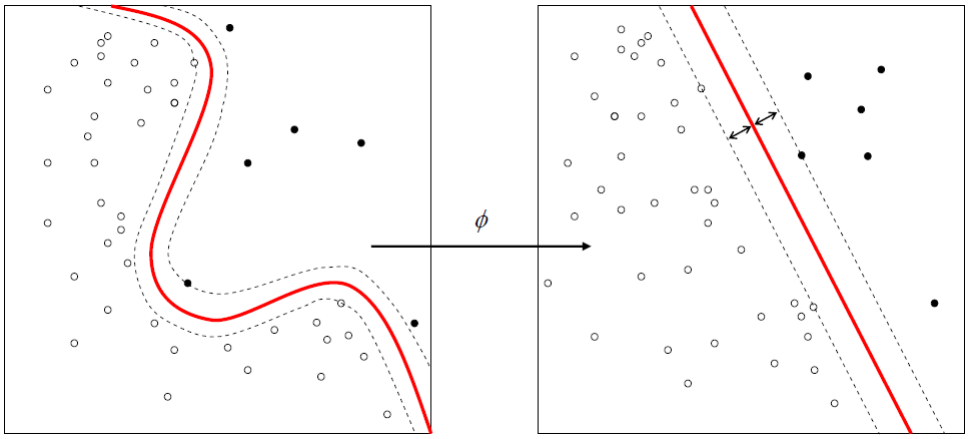


Fig. 7. Example of the Kernel Trick

D.5.3 Implementation The IDS Linear SVC classifier was implemented using scikit-learn's: sklearn.svm.LinearSVC class[22].

The IDS SVC classifier was implemented using scikit-learn's: sklearn.svm.SVC class[23], with a default RBF kernel.

[22] http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html
[23] http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

תקציר

למידה חישובית היא דרך מקובלת להוסיף למערכות לזיהוי קוד זדוני יכולת להתמודד עם קוד בו לא נתקלו בעבר. עם-זאת, במידה ולתוקף יש ידע על המסווג של המערכת – הוא יכול לשנות את הקוד הזדוני כך שלא יזוהה על-ידי המסווג.

בתזה הזו בנינו מערכת לזיהוי קוד זדוני, המבוססת על אלגוריתמי סיווג שונים של למידה חישובית, לפי קריאות המערכת המתבצעות על-ידי הקוד המסווג בזמן ריצתו. המחקר שלנו מראה ש- Random Forest הוא האלגוריתם בעל הביצועים הטובים ביותר.

לאחר מכן, הצגנו אלגוריתם הסוואה, שיוצר טרנספורמציה של רצף קריאות מערכת המזוהה כזדוני על-ידי מסווג מסוג עץ החלטה לכזה שמסווג כתמים, ללא אובדן הפונקציונליות הזדונית של הקוד. הדבר נעשה על-ידי הוספת קריאות מערכת עם פרמטרים לא ולידיים, שאין להן השפעה על פונקציונליות הקוד – אבל יש להן השפעה על מסלול הסיווג של הקוד בעץ. הדגמנו איך התוקף משתמש באלגוריתם כדי לשנות קוד זדוני לקוד זדוני שאינו מזוהה ע"י המערכת.

בחנו את ההשפעה של עדכון ה-training set של מסווג מערכת הזיהוי על אלגוריתם ההסוואה שיוצר בהינתן המסווג המקורי.

הצגנו מספר טרנספורמציות לקלט של מערכת הזיהוי (קריאות מערכת במקום ספציפי ברצף) בשביל להקשות על אלגוריתם ההסוואה. ראינו גם איך אפשר להתגבר עליהן באמצעות אלגוריתם הסוואה שתוכנן מולן.

הרחבנו את אלגוריתם ההסוואה לטיפול במסווג מסוג Random Forest, שזוהה כיעיל ביותר. לאלגוריתם שלנו 100% יעילות עבור כל המסווגים וטרנספורמציות הקלט בהינתן ידע מלא של התוקף עליהם.

לבסוף, הראנו שידע חלקי על ה-training set מספיק למימוש אלגוריתם הסוואה שעובד בהסתברות גבוהה.

גרסה מצומצמת של המחקר התפרסמה בכנסים WISTP 2016 ו-NSS 2016. גרסה מורחבת יותר תתפרסם בז'ורנל Concurrency and Computation: Practice and Experience.

המחקר שלנו מראה ששימוש באלגוריתם למידה חישובית עם training set גדול אינו פתרון קסם להתקפות: על מומחי אבטחה לקחת בחשבון שהמערכת עשויה להיות מושפעת מאלגוריתם הסוואה הדומה לאלו שהצגנו – ולנקוט בצעדי מנע כגון עדכון ה-training set באופן תדיר או שימוש בטרנספורמציות הקלט שהצגנו.

תוכן עניינים

תודות

ברצוני להודות לפרופ' גודס, המנחה שלי, שהשקיע שעות רבות בביקורת העבודה, במתן עצות שימושיות ובהשקעת זמן יוצאת דופן בהנחיה ובמחקר.

כמו-כן, אני מודה לאיתן מנחם מאוניברסיטת בן-גוריון על שאפשר לי להשתמש במאגר התוכנות התמימות שצבר בתזה שלי.

האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי המחשב

שיפורים בטכניקות להסוואה וגילוי של קוד זדוני

עבודת תזה זו הוגשה כחלק מהדרישות לקבלת תואר
"מוסמך למדעים" .M.Sc במדעי המחשב
באוניברסיטה הפתוחה
החטיבה למדעי המחשב

על ידי:

ישי רוזנברג

העבודה הוכנה בהדרכתו של פרופ' אהוד גודס

נובמבר 2016