FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

PPJ – A polymorphic Runtime Packer for Java

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Andreas Johannes Bernauer, BSc
Matrikelnummer 0627051

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: PD Dr. Edgar Weippl

Wien, 24.07.2011

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Technische Universität Wien

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

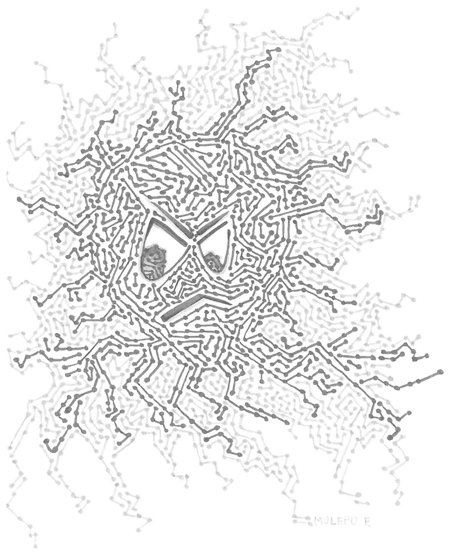
Wien, am 24. Juli 2011

Andreas Johannes Bernauer, Bsc

Contents

Vorwort

Was schreibt man in einem Vorwort für eine Arbeit, die einen Lebensabschnitt beschließt, an der man fast zu lange gearbeitet hat und schließlich aufgrund kaum zu bändigenden Interesses seitens der Kolleginnen und Kollegen kaum in der Bibliothek verfügbar sein wird? Man schreibt es zuallererst mal auf Deutsch; wenn man sich schon die Mühe macht, seine Arbeit auf Englisch zu schreiben, sollte man wenigstens das Vorwort auf Deutsch schreiben können. Jetzt, da es fast vorbei ist, bin ich zuallererst mal froh und dankbar. Auch, wenn es etwas kitschig klingt, finde ich doch, dass man hier ein paar Leutchens auflisten sollte, die einem wichtig sind und die es verdient haben, hier aufzuscheinen. Zum einen bin ich dankbar für meine Eltern und für deren Unterstützung in jeglicher Hinsicht. Es ist nicht selbstverständlich, solange von den Eltern finanziell unterstützt zu werden, sodass man seine Träume verwirklichen und Talente ausbilden kann. Ein herzliches *Dankeschön* an dieser Stelle für meine Eltern. Auf der anderen Seite ist da meine Freundin Julia, die mich nicht minder unterstützt hat und mir Mut zugesprochen hat, wenn es einmal Probleme gab und die sowieso immer für mich da ist. Zwei weitere Personen, denen ich danken möchte sind Niamh, die sich die Mühe gemacht hat, meine Arbeit auf Englisch zu korrigieren und Uli, der sich immer Zeit genommen hat, mich mit seinem Wissen zu beraten. Zu guter Letzt möchte ich an dieser Stelle noch meine übrigen Freunde erwähnen, die ich hier nicht beim Namen nenne, weil ich sicher irgendeinen davon vergessen würde. Ich kann mich glücklich schätzen, solche Freunde zu haben, auf die man sich immer verlassen kann und die einem zur Seite stehen.



Kurzfassung

Üblicherweise nutzt schadhafte Software eine oder mehrere Techniken zum Verschleiern von ausführbaren Dateien, um den eigenen Code zu verstecken. Die Transformation in ein solches geschütztes Programm nennt man „Packen“. Wird ein bestimmtes Programm mehrfach gepackt, so unterscheiden sich die einzelnen Instanzen des gepackten Programms zwar im inaktiven Zustand auf dem Speichermedium voneinander, im laufenden Zustand aber, wenn das Programm in den Speicher geladen und ausgeführt wird, werden für jede Instanz äquivalente Instruktionen ausgeführt. Es ist schwierig, für eine solche Art von Programmen Signaturen für statische Virenscanner zu erstellen, da die Eindeutigkeit nicht mehr mithilfe vom Programmcode auf dem Speichermedium ermittelt werden kann, sondern vorrangig durch den Code im Speicher oder durch das dynamische Verhalten gegeben ist. Diese Arbeit gibt einen Einblick in aktuelle Verschleierungstechniken und deren Gegenmaßnahmen. Weiters wird gezeigt, dass effektive Verschleierungstechnologien, die auf nativen Programmcode angewendet werden, auch auf die Java-Umgebung übertragen werden können. Da Java-basierte Plattformen und Frameworks einen immer größeren Anteil an etablierten Technologien einnehmen, wird es auch in diesem Bereich immer wichtiger, sich mit Schadsoftware und auch deren Verschleierung auseinanderzusetzen.

Neben einer Zusammenfassung und Besprechung von aktuellen Verschleierungstechniken, wird im Rahmen dieser Arbeit ein neuartiges, Java-basiertes Verschleierungsprogramm, ein „Runtime-Packer“, vorgestellt. Dieser implementiert eine effektive Technik zum Verschleiern von Java-Programmen. Der Packer basiert auf polymorpher Verschleierung, die eine Hülle, einen „Wrapper“, um das originale Programm legt, und so dessen Eigenschaften vor Virenscannern versteckt.

Um die Effektivität unseres Packers zu testen, wurde ein Prototyp entwickelt. Als Ziel sollten die implementierten Techniken so geschaffen sein, dass sie die Ausführungsgeschwindigkeit des gepackten Programms nicht signifikant verlangsamen. Trotzdem soll eine signatur-basierte Überprüfung für ein gepacktes Programm sehr schwierig werden oder sich die Analyse desselben durch einen menschlichen „Reverse Engineer“ erheblich zeitaufwändig gestalten. Letztendlich soll der Packer die Funktionalität des originalen Programms nicht verändern.

In einer Evaluierungsphase konnte gezeigt werden, dass unser Packer alle drei dieser Anforderungen erfüllt. Die Effektivität des Packers wurde mit Hilfe eines definierten Schemas evaluiert, welches den Grad der Verschleierung, die Ausführungsgeschwindingkeit und die Bewahrung der originalen Funktionalität berücksichtigt.

Abstract

Malware utilizes one or more obfuscation techniques to camouflage executables in order to conceal their own code. Thus in its dormant state on disk the program's code may vary for every copy, however when executed it carries out equivalent instructions to achieve its goal. It is difficult to generate signatures for this kind of malware, as its unambiguousness cannot be measured anymore by its provided code on disk but rather by its code in memory during runtime and by its dynamic behavior. This work gives an insight into current obfuscation techniques and their counteractive measures. Furthermore it is shown that effective obfuscation techniques that currently work in native machines can be applied to the Java environment. As Java-based platforms have gained increased market share, it has become more and more important to deal with obfuscation also in this sector.

Along with a summarization and a review of state-of-the-art obfuscation techniques, in line with this work a novel obfuscator in the form of a runtime packer is introduced which implements an effective obfuscation technique for the Java language. It is based on polymorphic obfuscation which puts a wrapper around an arbitrary executable in order to hide its potentially malicious nature from static malware scanners.

In order to test the effectiveness of our packer a prototype was developed. The techniques are expected to not significantly slow down the execution speed of the obfuscated program in comparison to its unobfuscated counterpart. Still, it has to yield excellent effectiveness, which means that an automated signature-based detection should become difficult and human reverse engineers should be impeded considerably with their work once a program is packed. Last but not least the packer should not alter the original functionality of the packed program.

In a final evaluation phase it could be demonstrated that our packer fulfills all three of the stated conditions. The effectiveness of our packer is evaluated against a defined measurement scheme taking into account the degree of obfuscation, execution performance and the preservation of functionality.

1 Introduction

While in the scientific area the dissemination of knowledge could be described as the foundation to progress, in other areas the desire exists to hinder this propagation for various reasons. A computer program can represent the manifestation of an idea which may hold some value that ought to be protected from others. In the case of an economical value a rival could incorporate this value in order to compete with a similar commodity. In other cases this value constitutes power over somebody or something else while a competitor tries to take over this power or to abolish it. In the specific sense a virus author—or an author of malware, in general—aims to gain power by taking control over an alien system, while the anti-virus software tries to protect this system by taking this power away from the malware. From the perspective of malware-writers it is desirable to protect their creations—their malware computer programs—from others who try to figure out how the vermin works, how to detect it and finally how to remove it.

Although there is automated analysis, yet, it is mainly a human's work to analyze unknown malicious binaries, especially if a deeper understanding of an unknown binary is required. While it is already difficult for humans to understand programs in machine code form (cf. 2.3) there are ways to artificially complicate programs in order to make them hard to understand and to analyze, for both humans and machines. Obfuscators take on this task and can take the shape of so-called *runtime-packers*. Such a packer takes a given program as input and puts a wrapper around it which acts as a protection layer. During runtime this wrapper (or stub) extracts the original program automatically and transparently to the user and finally hands over control to the packed program which performs its actions as before. The user doesn't notice that the program was packed. An attacker, however, is intended to have a hard time dissecting the file.

This thesis deals with the questions 'What is obfuscation in relation to computer programs?', 'Why would anyone try to apply obfuscation to a program?', 'How can it be done?' and 'How can a concrete implementation of such a technique look like and how would it perform?'. On the one hand one contribution of this thesis is a summarization of literature about program obfuscation; it covers definitions about obfuscation, ways to measure it and challenges its usefulness. Furthermore obfuscation techniques are described in-detail, so are the according counter-measures. On the other hand a novel java-based runtime-packer, called 'PPJ', was developed in line of this work. Together with its evaluation against defined measuring methods this forms the other and main contributions of this thesis.

It is divided into several chapters; the next one, chapter 2 concentrates on program obfuscation in general, what types and techniques there are, how they work and how they can be mitigated. In chapter 3 the concept of this packer is described and details about its implementation are given. The following chapter 4 is about the evaluation of PPJ.

Here details regarding several aspects of the packer's performance are stated, regarding execution time performance, preservation of functionality and the degree of obfuscation what could be achieved. The final chapter 5 gives a summarization of this paper and its results and offers an outlook on future developments.

2 Program Obfuscation

Runtime-packers are a specialized form of program obfuscators that apply some sort of obfuscation to a given program in an automated[1] way. In order to understand the details (cf. section 2.2.2), initially general aspects of program obfuscators and then the act of applying obfuscation to a program itself are discussed in the following.

A "program obfuscator is a compiler that takes a program as input and produces another program as output". Alongside this definition, Barak also defines three conditions that an obfuscator has to fulfil [2]:

- **Functionality:** "The obfuscated program should have the same functionality (that is, input/output behavior) as the input program." The way it achieves its goal can differ, but finally both programs should yield the same output, if the same input is given.
- **Efficiency:** A standard compiler optimizes a program in a way so that it will take as little time as possible to execute. When this program is obfuscated it is likely that some overhead will slow down its execution speed. The goal is to keep this overhead low, so that the program will not run noticeably—say exponentially—slower.
- **Obfuscation:** Obfuscation is a generic term for several techniques to render program code preferably hard or at best impossible to understand by humans. Later in this text we will see that this is not always possible. We say if a cracker manages to recover the original source code of an obfuscated program P, then the obfuscator *completely failed* on program P.

A different, more refined view that focuses more on the properties of the applied obfuscation and is intended to measure the "quality of an obfuscating transformation" is given by Collberg et. al. in [10].

- **Potency:** the amount of obfuscation (i. e. complexity) added. Collberg et. al. consult complexity metrics proposed by McCabe [24] and Harrison et. at. [18] that identify the number of predicates and the number of nesting levels that conditional branches and loops introduce as indication for some code's degree of obfuscation.
- **Resilience:** the degree of difficulty to remove the applied obscurity by an automated deobfuscator. Two factors contribute to the total scale of resilience: firstly

[1] Here 'automated' means *the absence of any human intervention* in the process of transforming an input program into an obfuscated version of itself, with the exception of possible pervious configuration steps of this very transformation process.

the *programing effort* to create a deobfuscator that is capable of reducing the applied obfuscation's potency and secondly the *deobfuscator effort* measured in time and space needed for its execution.

- **Stealth:** how well the code responsible for obfuscation is integrated into the rest of the code i. e. how much it differs from the original code and is therefore noticable to a human.
- **Cost:** how much overhead is added to the program that might lead to increased execution time or allocated space.

In the following an introduction on obfuscation is given; how it can be used and what its potential and its constraints are.

2.1 Use of Program Obfuscation

In this section the usage of program obfuscation is highlighted. Examples for applications and finally a critical discussion on security through obscurity are given.

2.1.1 Applications

Obfuscation techniques apply to several kinds of languages, from low-level to high-level, from assembly language to Perl. However, as there are various means to conceal code by exploiting the specifics of a given programming language, there are also miscellaneous reasons as to why one wonders about obfuscating program code.

Impeding A/V Software

Typically, static malware scanners rely on program signatures to identify malware. In the view of a malware author it is reasonable to obfuscate malicious program code in order to complicate the work of a human malware analyst or to thwart analyzing applications that take on to automatically analyze the malware's code and to generate signatures. Secondly, obfuscation can help to beat malware scanners itself so that an obfuscated malicious program can execute without begin offended by a virus scanner.

Brainteasers

Another more peculiar reason is the pleasure that writing and reading obfuscated programs gives to some programers. Hard-to-read source code is appreciated as a challenging riddle that sharpens programing skills and extends the knowledge of the researched language. There are even several contests in which witty participants can earn rewards for their most creative obfuscated code, whereas there exist contests for many languages, especially for C[2], Perl, Ruby[3] or PostScript. Program 2.1 shows a typical example of obfuscated C code that calculates π by taking its special formatting into account. Programs

[2]IOCCC: http://www.de.ioccc.org/main.html
[3]IORCC: http://iorcc.blogspot.com/

```
#define _ -F<00||--F-OO--;
int F=00,OO=00;main(){F_OO();printf("%1.3f\n",4.*-F/OO/OO);}F_OO()
{
            _-_-_-_
       _-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
_-_-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
 _-_-_-_-_-_-_-_-_-_-_-_-_-_-_
  _-_-_-_-_-_-_-_-_-_-_-_-_-_
    _-_-_-_-_-_-_-_-_-_-_-_
        _-_-_-_-_-_-_-_
            _-_-_-_
}
```

Program 2.1: Obfuscated C program calculating π, [42]

of this kind often use weird formatting, preprocessor redefinitions or self-modifying code to make their code more difficult to understand.

Homomorphic Encryption

Barak et al. [3] show some other more serious and no less interesting fields of applications for program obfuscation: by finding an obfuscation technique that will not fail, one could think of an implementation for *homomorphic encryption*. Homomorphic encryption is an encryption scheme in which it is possible to apply some operation $\otimes$ on two variables a' and b' that have previously been encrypted by function E, where $a' = E(a)$, $b' = E(b)$ and the inverse function to E is D. There also exists a possibly different operation $\oplus$ that issues the same result r for $r = a \oplus b$ and $r = a' \otimes b'$. By implementing function D in an obfuscated way, it would be possible to safely distribute D together with a' and b', where $a' \otimes b' \mathrel{\widehat{=}} D(a') \oplus D(b')$. As an example homomorphic encryption makes it allowable to add two encrypted values yielding a specific outcome that is equal to the outcome of the addition of the two unencrypted versions of those values.

Under common circumstances the problem with the process of ‘decrypting—applying the operation—re-encrypting’ without obfuscation is that normally every en-/de-cryption algorithm depends on a key that not everyone should have access to. However, one can think of a situation where encrypted values are passed on to a set of people. They should have the right to apply a specific operation on those values, but should not be able to freely decrypt them and therefore realize the plain text. While on the one hand the key is needed to decrypt the values in order to perform the operation, the

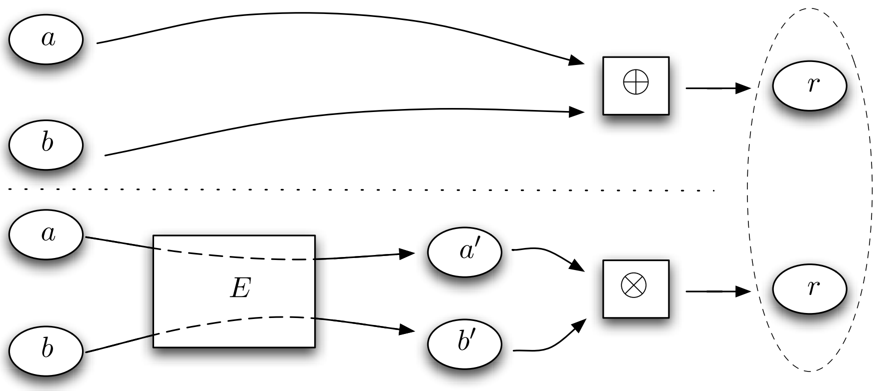


Figure 2.1: The diagram shows how the application of homomorphic encryption is intended to work. A binary operation $\otimes$ allows to apply encrypted values a' and b' and issues the same result r as operation $\oplus$ that is applied to the unencrypted values a and b.

decryption key must not be passed on with the encrypted values so that people can use it to decrypt the values without restriction. The previously described implementation of homomorphic encryption solves this problem by making the process, consisting of decrypting, performing the operation and re-encrypting atomic and distributing the key along with the encrypted values, in an obfuscated and therefore unaccessible way.

Public/Private Key Encryption

As a second example of an application that takes advantage of obfuscated code, Barak et al. explain how a novel public/private key encryption scheme that entirely relies on obfuscation can be created by *obfuscating a symmetric key encryption scheme.* The security of conventional public/private key encryption schemes relies on trapdoor functions which make use of mathematical problems that are believed to be difficult to be solved. A trapdoor function can be utilized to easily compute some cipher text c by encrypting x using a publicly available key k_{pub}. However, it takes disproportionally much more computing power and time to re-calculate x out of c without the knowledge of some secret key k_{prv}. It is not known whether finding the inverse of c is indeed difficult or if some easy solution has just not been found yet.

By applying obfuscation, the implementation of a symmetric encryption function $E_{k_{sym}}$ depending on a specific key k_{sym} could be concealed. In doing so $E_{k_{sym}}$ can be made public so that everyone can encrypt arbitrary data without ever realizing the key and finally obtain cipher text c. Only those people with knowledge of k_{sym} would be able to decrypt c and get back the original data x. Others that have access to $E_{k_{sym}}$ only are not able to extract k_{sym} as it is embedded in the implementation's code which is obfuscated in turn. Figure 2.2 graphically demonstrates this.

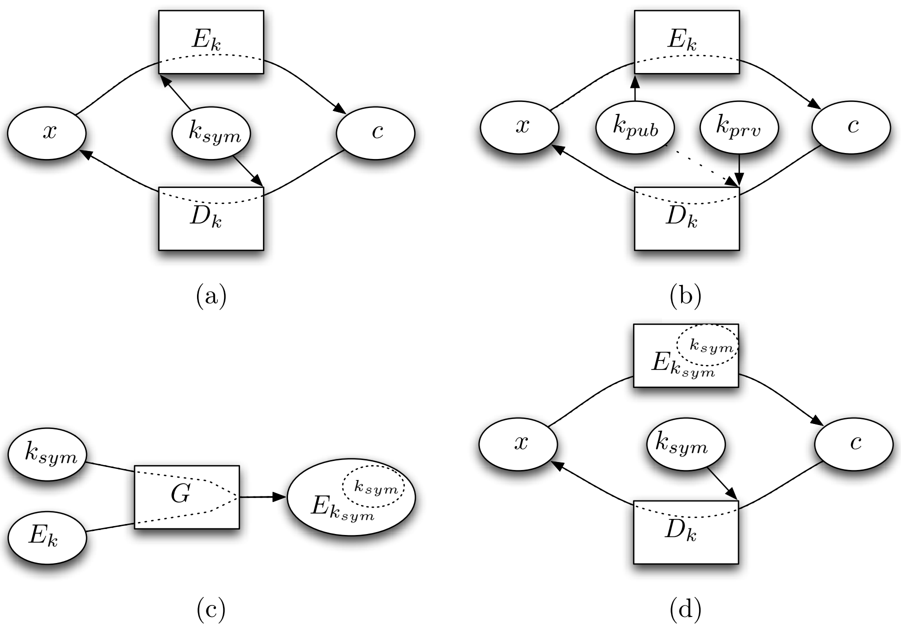


Figure 2.2: Whereas symmetric en-/decryption (a) requires one single key for both, in a classic public/private encryption scheme (b) there exists a key-pair. One part, the public key k_{pub}, is responsible for data encryption and should be distributed freely so that anyone can encrypt data x to send it to the owner of the key-pair in a secure way. It is not possible to use k_{pub} for decrypting the cipher text c in reasonably short time. The second part, the private key k_{prv}, must be kept secret as this key allows the decryption of data that was previously encrypted with k_{pub}. An obfuscation generator G (c) is used to generate an encryption scheme $E_{k_{sym}}$ (d) that has a symmetric key k_{sym} secretly embedded in itself and therefore does not take anything but the plain text x as input. The inverse function D_k can decrypt c, however this time the secret key must be provided.

Digital Rights Management

In a scenario described by Collberg and Nagra [8], a user should only be able to use some digital (multimedia) content in a limited way. For example, the user has only paid for the right to render some content, thus he or she is allowed to listen to a set of audio files. The user is not allowed and should not be able to distribute them to a third person either. On a technical level there is only a very small gap between copying a digital file and opening it for playback. Generally one can assume that if someone is able to render a file he or she is also able to copy it arbitrarily if there are no further precautionary technical measures. Cryptography can be of assistence in this case as it makes it possible to encrypt the content which cannot be decrypted (and therefore opened and played) without the key.

In a possible scenario every time a file that has to be opened for playback, the media player requests the key in some secure way from a server of the music-selling party. It makes no sense for the consumer to give away the encrypted music file because an authentication service grants access only to the original buyer who is tied to the personalized media player. The crux of this system is that no playback is possible if the media player is offline and therefore has no access to the key distribution server. So, what designers of an DRM system usually do is to store the key in an obfuscated way on the customer's machine. If obfuscation worked flawlessly in real-world scenarios this would be the solution to the problem, but there are several weak points an attacker can focus on: It is generally not a good idea to store the key together with the cipher text. Even if the key is well-hidden the past has shown that in reality, it is only a question of time until either the key is found or even the key-embedding system is cracked and therefore any key can be extracted for a specific user that uses this system. Another possible attack one could think of is to apply a hook into the system that diverts the digital media stream into an unprotected file which can be passed on without restriction afterwards. Last but not least there is still the old technique of digitally recording the analogue output which will admittedly yield lower audio quality (dependent on the equipment used) in comparison to the techniques described previously.

Anti Reverse Engineering

All the previous use-cases benefit from the fact that obfuscated program code is hard to understand, however not only cryptographers and minds that want to be challenged are interested in obfuscation. By far the most cases where obfuscation is used is when developers try to protect their software from being reverse-engineered. In fact this seems to be the main reason why obfuscation techniques are continously further developed. Commercial software is usually distributed as a compilation in the form of machine-code, which is very hard for humans to understand and already provides obfuscation to a certain degree. Normally it is not an enormously difficult task for experts to tamper with it: further obfuscation techniques are required.

According to [12], reverse engineering "is the process of extracting the knowledge or design blueprints from anything man-made". When technology became part of every-day life people began to take apart things to try to find out how they work. In opposition to this hobbyism-driven reverse engineering, in industry experts conduct reverse engineering to gain design information that is not documented. In this case, the original producer may have tried to hide this information or it may have been lost over time.

Without dwelling on the legal issues of this topic reversing is done for several reasons in the software business, cf. [12, page 8].

Interoperability with proprietary software: Though the documentation for an API of some proprietary software is available, it can happen that still not all details are clear for the developer that wants to use it. An option is to get in contact with the manufacturer who can clarify the ambiguity. However if the developer holds some powerful reverse engineering skills it is often easier and more time efficient to resolve the problem by having a look at the implementation itself.

Creating competing software: Reverse engineering mostly takes place when a company tries to catch up with its competitor by placing a similar product on the market. Apart from other businesses, surprisingly in the software industry reverse engineering is not the usual procedure to gain know-how about new technology that has made a competitor successful. In nearly any circumstance distributed software is far too complex and reversing the entire application would be too costly. Re-implementing the product from scratch and trying to make it even better is usually the way things are done. There are exceptional cases when it is very expensive to develop a specific design or a sophisticated algorithm. In such cases the situation is different and the costs for extracting this algorithm by reverse engineering are lower than for developing an equal one for one's own. Again the majority of an application built around this re-implemented reversed code will still be developed independently without the help of reverse engineering.

Cracking software: Developers sometimes equip their (commercial) application with mechanisms to restrict the usage of the software in some way. Mostly the functionality is limited to some degree. This restriction is deactivated if the user enters a serial number that he or she has obtained from the manufacturer after they have received the stated payment demanded of the user. Crackers use reverse engineering techniques to either circumvent the built-in verification which checks if the user has already paid for the software, or they even extract the verification algorithm itself to build a serial number generator that will generate a valid verification code whenever needed. The latter is the cleanest solution (and the most fatal one from the manufacturer's view) as it requires no alteration of the program's binaries but simply yields a valid serial number that the manufacturer would normally distribute to satisfy the application's verification check. After a certain software is cracked, other people can easily use this software without constraints and without paying the company that originally intended to make money with the product.

2.1.2 Criticism

From a scientific viewpoint the employment of program obfuscation is not always unobjectionable. In the following it is explained why.

Well-Defined Security

The problem with obfuscation is that it is generally not well-defined. This means there is no distinct mathematical definition of which requirements a certain type of obfuscation has to fulfill exactly, what class of programs these requirements apply to and what the limits are of this specific obfuscation.

Note the difference: *well-defined security* refers to a concept, for example, that is a public/private crypto system with all its properties or a collision-resistant hash-function. In opposition to this, *proven security* refers to a specific implementation of this concept for which exists mathematical proof that this very implementation satisfies the requirements that are given by the definition. Therefore, a specific implementation cannot be mathematically proven to work accurately if the corresponding mathematical definition is missing. “[H]aving a well-defined cryptographic concept is a necessary condition for proven security”, as Barak puts it in [2].

The importance of well-definition must not be underestimated: it lays the mathematical foundation that allows experts to prove that a specific implementation of some code works accurately; that it cannot be broken if certain requirements are met. Writing secure code is already a non-trivial task, so it is essential that developers can access strong and secure algorithms which do not have any hidden weaknesses or subtleties that cancel its strength under any circumstances.

Barak calls this kind of undefined security “fuzzy security”. As an example an inventor of a novel cryptographic algorithm claims his algorithm to be secure without any mathematical demonstration. While people begin to establish it in productive environments, a cracker finds a vulnerability, whereupon the algorithm is tweaked with the conjecture that the new version makes the algorithm immune to the attack. It is hoped that there are no more other vulnerabilities hidden or if so, that nobody will find them. Considering the fact that obuscation techniques lack well-definition, the previously described course of action is a common approach when it comes to obfuscation. Also, a tactical decision is to try to hide the implementation details to make it even harder for attackers to find a loop hole (see also section 2.1.2). Under these circumstances no security assurance can be given.

It is likely that an implementation of some secure application will always be faulty but at least the theoretical concepts of an algorithm should be reliable and securely strong. Certainly an implementation can be tested by convential software-testing means, but this can only assure that a system will work within the realm of tested parameters. A cracker will possibly try to break the system by giving some input that the tester did not check. Compared to mathematically proven cryptographic standard algorithms, like the RSA cryptosystem, which for example relies on the definition of digital signatures, software testing can only give a very weak verification that some piece of code actually works.

Barak et al. tried to find a formal mathematical definition for software obfuscators, however this turned out to be a very difficult task. On the one hand the potential definition should be weak enough so that one cannot find any counterexamples and on the other hand it should be “strong enough to provide some meaningful sense of security”. The weakest obvious definition the author could think of was:

> “A compiler satisfying the functionality and efficency conditions is an obfuscator if for every program P that is hard to learn from its input/output behavior, a hacker can not reconstruct the source code of P from the obfuscated version of P.”

Still counterexamples, such as programs that do not meet the requirements mentioned before, were found for this definition. A possible solution by making the definition even weaker, is to define a subclass of programs that includes those kind of ‘natural’ programs that developers usually want to obfuscate in practise and excludes those counterexamples that will contradict the definition. However, this problem remains unresolved. Hereby the difficulty was to distinguish between the counterexamples and the ‘natural’ programs. Also, an open question remains as to how a user can easily be able to check if a specific program can be obfuscated securely or not. See [3] and [2] for a more elaborate discussion on this topic.

Security Through Obscurity

The vagueness of the security of obfuscation is related to the discussion on the reasonability of security through obscurity. Security through obscurity is the notion that the internal design of a system e. g. used algorithms, implementation, etc. should be kept secret in order to establish or enhance security. Although the system may have security flaws it is assumed that because of the concealed internals an attacker won’t find those vulnerabilities and therefore won’t be able to exploit them. Contrary to this is *kerckhoffs’ principle* that postulates an open system which everyone, even the ‘enemy’, can evaluate. The security merely relies on a key which has to be kept secret. The advantage of a laid-open design is that other experts apart from the original designers can test the system for vulnerabilities and help to make it stronger. Nowadays security through obscurity is generally deprecated by renowned security experts as a bad design principle. Bruce Schneier writes about it in [35, page 8]:

> “If you believe that keeping the algorithm’s insides secret improves the security of your cryptosystem more than letting the academic community analyze it, you’re wrong. And if you think that someone won’t disassemble your code and reverse-engineer your algorithm, you’re naïve.”

Schneier substantiates his statement by referencing the RC4 algorithm, whose source code was originally kept as a trade secret until 1994 when it was anonymously published to the Cypherpunks mailing list, from where it found its way to other newsgroups and FTP servers.

On the other side people often have a more relaxed view on security through obscurity. In [4] Beale points out that there are cases when obscurity will very likely slow down an attacker or push him to become more noticeable. As an example, if an internal web server is not bound to its well-known port 80 but listens to some uncommon port like 31337 an attacker has to run a portscan to find the server’s web service before he or she can attack it. Another easy but effective method of system hardening is making banners[4] less verbose or changing their text to reflect inaccurate values. No doubt, changing common ports is no substitution for a secure authentication system to prevent unauthorized access.

[4]When a client connects to a server it will reveal itself by giving the client information about the service type, the vendor, the implementation name and the version number. This information is communicated via a ‘banner’ which is embedded in the protocol that the client and the server use to communicate with each other. A banner as part of an HTTP-response can look like this: `Server: Apache/2.0.55 (Debian) PHP/4.4.2-1.1`. This gives attackers valuable information about the system and makes it easy to find out which known vulnerabilities the system is prone to. There is no need for these values to be thorough, so one can change or remove these chatty banners without demur.

Also changing banners won't restrain an attacker from only trying to run a dedicated attack on your service or finding out details about your service over some other channel (for more examples supporting the application of security through obscurity read [36]). However these are all means to reduce the attack-surface and make your system harder to compromise; unexperienced crackers or automated attacks in particular can be easily fooled. In this manner a good proportion of 'background noise'-attacks can be repelled. As the attacker is often forced to run a more wide-spread and 'louder' attack, purposeful security through obscurity reduces the likelihood that the attacker won't get noticed as an intrusion detection system could become aware of the attack. In this case precious data can also be collected by tracking the attacker's behavior.

Yet Beale agrees that security which solely relies on obscurity is by far not enough to cover all security needs nowadays. However in many cases it works well with solid security measures. Security by obscurity can help to reduce indiscriminate attacks but an attacker who really is determined to break into the system will find a way to circumvent defense mechanisms driven by security through obscurity sooner or later.

So, what exactly is the relationship between security through obscurity and program obfuscation? Both terms are sometimes used equally; in fact, they describe different concepts. Although it is not common practice, an algorithm for obfuscating program code could be made public for general review. As pointed out in [8], depending on a seed obfuscation schemes could be applied to executables. This process would yield diverse, obfuscated programs as the seed would determine how and where the obfuscation algorithms are applied to the input programs. However as shown before, it is currently not possible to raise any obfuscation algorithm into the domain of proven security. So, in practice developers rely on security through obscurity to firstly make sure nobody finds out how the obfuscation algorithm works and secondly to enforce the anti-reversing mechanisms. In a nutshell, theoretically there is no need to use security through obscurity when it comes to program obfuscation, but as long there exists no standardized algorithms it is often used to make it harder for reversers to dissect the internals of a protected executable.

2.2 Obfuscation in Executables

Obfuscation in executables is probably the most significant reason why developing new techniques in this realm is interesting to a lot of people nowadays. It can be seen as a powerful tool that not only developers make use of in order to protect valuable algorithms and designs from being stolen but it is also there to help in hiding the internals of malware and make them stealthier by decreasing the probability of being detected by antivirus software. While we discussed theoretical aspects of obfuscation in the last section, the next section gives an overview of obfuscation techniques in executables.

2.2.1 Poly-/Metamorphism

Traditionally an antivirus software[5] searches for malware by comparing the bytecode of a suspicious program with a set of unique signatures. These signatures are stored in a library which is continuously maintained to keep it up to date. Whenever a new kind of malware is encountered, a certain sequence of bytes needs to be found that uniquely identifies the malware which is then utilized to create a signature which is entered into the database so that scanners can use it to find the corresponding malware on infected computers. Poly- and metamorphic malware tries to fool antivirus software by changing its bytecode while preserving its original functionality (cf. [12] and [39]).

Polymorphic Code

Early polymorphic viruses use encryption to change their shape. A constant header is responsible for decrypting the body in memory before it can execute its actual function. Although the virus is able to change its shape, it is still possible to extract a signature from it by examining the constant decryption stub which never changes. The second problem (from the virus author's viewpoint) is that at a particular moment the whole decrypted—and constant—virus body is in memory which would also make it easy to apply a signature-based search.

The following generation of polymorphic viruses aims to obliterate one weak point by introducing decryption stubs that would look different for every new generation. Those *oligomorphic viruses*, as they are also called, can change their decryption header's code in finite ways and nevertheless keep their original function. A simple method described in [12, page 282] is to rotate or randomize the use of registers:

Commonly there is no rule about which general-purpose registers, like `EAX` or `EDX`, have to be used for a set of operations.

```
00403448        8B45 D4        MOV EAX,[EBP-2C]
0040344B        8945 D8        MOV [EBP-28],EAX
0040344E        8B45 DC        MOV EAX,[EBP-24]
00403451        3345 D4        XOR EAX,[EBP-2C]
00403454        8945 DC        MOV [EBP-24],EAX
```

The instructions at `00403448` and `0040344E` load certain values into register `EAX` that is used in the particular instruction below. Here `EAX` can be exchanged by a different unused register which leads into some different bytecode. This could fool an antivirus engine because the recorded signature will not match the virus' code anymore.

```
00403448        8B5F D4        MOV EBX,[EDI-2C]
0040344B        895F D8        MOV [EDI-28],EBX
0040344E        8B4F DC        MOV ECX,[EDI-24]
00403451        334F D4        XOR ECX,[EDI-2C]
00403454        894F DC        MOV [EDI-24],ECX
```

[5]Though the term 'antivirus software' suggests the sole detection of viruses, in fact, antivirus solutions are usually full-featured security solutions that in addition to viruses are also able to detect other malware such as trojan horses, worms or spyware.

Above is a second version of the first code that uses different registers. As said before, the according bytecode also changes along with the registers in every instruction. Note that a certain register A is not just exchanged with register B, but the mutation algorithm also takes into account the register's content. In the original code at `0040344E` the value of `EAX` is overwritten by a new value of a different context. The old value is not needed anymore and `EAX` now handles the new value which gives us the opportunity to also use a different register for the successive, mutated code. This happens at `00403448` and `0040344E` in the second code block where first `EBX` and then `ECX` is used as a substitute for `EAX`.

A typical example for a second generation type of a polymorphic virus, Win95/Memorial, is capable of building 96 different stubs. Therefore once all possible shapes of the header have been recorded, signature-based detection still works, although due to the high number of different stubs, this is not a practical solution. The second approach, detecting the decrypted constant virus body in memory, is more popular and mainly used by antivirus software.

Finally, this design was superseeded by those polymorphic viruses that could create an endless number of different-shaped decryption headers which would decrypt its constant body by using several methods. With time virus authors applied further improvements like multi-layered encryption, anti-emulation techniques and Random Decryption Algorithm (RDA) which aimed to make unpacking, thus decrypting the virus body, even more difficult for antivirus engines. Here the virus does not store the key together with the encryption algorithm and the encrypted body in the host file, but omits the key, so that an antivirus scanner is not able to decrypt it easily. The only way to retrieve the original plain text is a brute-force attack. This is also true for the virus' decryption stub that in fact launches a brute-force attack on itself when executed. That may sound ineffective, however, while undetected the virus can take its time to find the right key and won't be bothered as it is dormant for some time. On some machines it may even fail to decrypt itself; in contrast to this though the antivirus software cannot afford to overlook the virus.[6]

According to [39] at the time when polymorphic viruses were more 'in the wild', strategies against those malicious viruses were dynamic decryption (which works as the virus' body is still constant once decrypted) or the application of code emulators that some antivirus producers had already implemented in their engines.

Metamorphic Code

The blind spot of polymorphic malware is its constant body, that has to be present somewhere in memory before the virus can start to execute its payload. Metamorphic viruses take the next step to an even stealthier type and try to eradicate the previously described problem. Techniques which conceal the decryption header in polymorphic viruses are improved to make them more sophisticated and are applied to the virus body. The result is a virus that lacks a decryption stub but has a complex metamorphic engine

[6]For more information on RDA see: http://www.awarenetwork.org/etc/alpha/?x=3.

```
--------        85C0            TEST EAX,EAX
;equals
--------        21C0            AND EAX,EAX
```

```
--------        33FF            XOR EDI,EDI
;equals
--------        BF 00000000     MOV EDI,0
```

```
--------        8D45 FC         LEA EAX,[EBP-4]
;equals
--------        8BC5            MOV EAX,EBP
--------        83E8 04         SUB EAX,4
```

Program 2.2: The first example shows the equivalence (i. e. the same processor flags are set) of `TEST` and `AND`. A `XOR` instruction can be used to reset a register by applying the binary `XOR` operator to this register twice; the `MOV` equivalent above does the same. A longer version of the `LEA` command that subtracts 4 from the value in `EBP` and stores it into `EAX` can be seen as a combination of `MOV` and `SUB` in the last example.

that drastically alters the whole virus for every successor it creates. It analyses its own code to find locations where it can apply one of its mutation techniques. In [12] Eilam describes several techniques as to how a metamorphic virus manages to change its code while keeping its original function.

Instruction and Register Selection: A possible method to give the code a new shape, is to exchange the registers that are used for certain instructions. Except for some that have a specific meaning (e. g. `EIP`), registers can be arbitrarily chosen to hold a value. This fact is utilized by metamorphic engines in order to change the opcode and make it harder for antivirus software to detect the virus. A related method is the substitution of an instruction by an equivalent one. The examples in program 2.2 compare two or more instructions which issue the same outcome but partially have a completely different opcode.

Function and Instruction Ordering: The order in which functions are stored in a binary does not matter, normally. A metamorphic engine can reorder the function's code, though it still has to adjust all jump references to the function of course. Also certain portions of instructions that are arranged one after another but do not depend on each other can be interchanged.

Reversing Conditions: Depending on some conditions a program can follow different branches which depict themselves as blocks of code in the binary. A typical example is that if two values are equal, the program executes a `JMP` instruction at the beginning of the according chunk of code that has to be processed in this case. In the other case the

```
00401355     85C0          TEST EAX,EAX
00401357     75 07         JNZ 00401360
00401359     8D03          LEA EAX,[EBX]
0040135C     FF00          INC [EAX]
0040135E     EB 05         JMP 00401366
00401360     8D06          LEA EAX,[ESI]
00401363     FF08          DEC [EAX]
00401365     50            PUSH EAX
00401366     ...           ...
```

```
00401355     85C0          TEST EAX,EAX
00401357     74 07         JZ 00401361
00401359     8D06          LEA EAX,[ESI]      ○
0040135C     FF08          DEC [EAX]          ○
0040135E     50            PUSH EAX           ○
0040135F     EB 05         JMP 00401366
00401361     8D03          LEA EAX,[EBX]      ●
00401364     FF00          INC [EAX]          ●
00401366     ...           ...
```

Program 2.3: Both samples of assembly code are equal whereas the second example has had its conditions reversed. Considering the first example at address `00401355` the program checks whether the value in `EAX` is 0 and in the next line it jumps to `00401360` if the check was negative. Therefore, some value stored in `ESI` will be decreased and then pushed onto the stack if the value in `EAX` was unequal to 0. In the other case the program will not jump but will just execute `00401359` where another value in `EBX` will be increased before it unconditionally jumps to address `00401365` where the program will continue with its work. The second example shows that instruction `JNZ` was changed to `JZ` which will cause to program to jump if the test was positive. The lines of both branches (marked with • and ◦ respectively) need to be exchanged in order to maintain the original program flow.

program just executes the next instruction (which represents the beginning of the other code block that is responsible for the case that the two values differ) and 'ignores' the `JMP` command.

An opportunity offers to re-arrange some code by reversing a condition, i. e. instead of checking whether some value is zero the program checks if the value is *not* zero. As the condition is inverted now without further intervention the wrong branch of code would be executed. Thus to preserve the original program flow the two code blocks get exchanged. This leads to a major shift of bytecode that can help to thwart antivirus software. For an example see program 2.3.

Garbage Insertion: By inserting so-called garbage code it is possible to alter the program code further. In general the aim of garbage code is to manipulate insignificant data that will not alter the function of the program but will bewilder human reversers and also antivirus software. `NOP` instructions which serve as a filler and do not perform anything, are just a very simple example of garbage insertion. One could think of pushing garbage data onto the stack and removing it again before an actual valid value below the garbage is needed again. Garbage insertion goes well with function and instruction ordering: besides reordering blocks of code, garbage can be inserted in between blocks whereas `JMP` instructions direct the program flow so that the garbage is avoided and the code is executed in the right order.

Code Integration: Péter Ször and Peter Ferrie describe a technique the virus W95.Zmist is known for [39]. So far the described obfuscation techniques only apply to the virus' own code, however, code that is capable of code integration may even modify the host's code by blending into the normal program flow. In terms of a virus this requires a huge amount of resources because the embedding process needs to disassemble the host's code, re-arrange chunks of code, therefore update references and finally metamorphically insert the virus' code and re-build the executable. The process requires the host machine to provide 32MB of RAM for the metamorphic engine; nowadays this portion might not seem much, however, in 2000 when the Zmist engine was disengaged this was a good deal of the overall amount of RAM a computer was equipped with.

2.2.2 Runtime Packing

A runtime packer is a specialized program obfuscator as described in section 2.1. In contrast to general obfuscators, whose definition does not specify on which level they operate, an important property of runtime packers is the transformation of code in memory during runtime. A classical runtime packer consists of transformed program code on the one hand and a stub on the other hand. The program code is protected in some way and often not directly executable, the stub, however, can be executed and is responsible to transform the protected code in memory to bring it into an executable form and finally handing over control to it. This makes runtime packers specialized obfuscators that conceal program code in a specific way. Although most runtime packers strive to satisfy all the conditions mentioned in section 2.1, dependent on the intended usage the emphasis varies on what requirements a specific type of runtime packer has to fulfill.

As already discussed in section 2.1.2 the security of obfuscation does not stand on such solid ground as cryptographic algorithms do. Despite this and probably also because of the lack of alternatives, program obfuscation is a popular means to complicate the disassembly of some programs. Although it would be desirable to be able to resort back to a method which features all the benefits of a strong cryptographic algorithm, for most users the more or less latest technology suffices and provides enough leads in this cat-and-mouse game between reversers and manufacturers of runtime packers. Following the principle of security through obscurity (cf. section 2.1.2) the application of runtime packers may still discourage the majority of attackers. Even though awareness of the fact that a single reverser is enough for some secure code to become available for everyone, people act true to the motto ' It's better securing code with insecure obfuscation than leaving the code entirely unprotected'.

It is interesting that the implementations of runtime packers share the same properties which make them program obfuscators, and may still completely differ from each other on a technical level. This can be observed on the basis of the packers' output in the form of packed binaries that have different, partially oppositional attributes. Due to accretion of new and different techniques over time and due to changes in the purpose of the runtime packers' application, the notion of the term 'runtime packer' became ambiguous and blurry. This led to a fragmentation of various implementations into subgroups that are highlighted in the following.

Types of Runtime Packers

Compressor: A compressor is a type of runtime packer which compresses input executables and adds a stub to the packed payload that is responsible for extracting it. When a packed executable is loaded by the operating system, the stub will dynamically extract the original executable in RAM and once its work is done, will hand over control to the unpacked code which then will carry out its original function. This method saves space on mass storage media and comfortably integrates the procedure of extracting into the normal program flow without the user's intervention.

Clearly the aim is firstly to maintain the original code's functionality and secondly not to impair the execution speed which is why a great deal of brain power is dedicated to implementing algorithms with a focus on decompression speed. Nowadays, as storage space has become and has remained cheap, in exceptional cases only saving disk space is still an important matter. This very fact and the name runtime 'packer' suggest that compressors are what the term 'runtime packer' originally was related to. Finally, people realized that compressors have the potential to obfuscate executables whatever the purpose (concealing malware or protecting intellectual property) it may eventually serve at last. This shifted its purpose from being a tool for saving disk space to being a program obfuscator.

Examples of popular compressors are UPX[7], MEW, or ASPack[8]. UPX is quite prevalent as it is open source, continuously maintained and improved. MEW is not commercially distributed but rather has its origin in the internet's underground.

Cryptor: As the name denotes, a crypter encrypts its executable payload in order to hide the content. The technique is reminiscent of polymorphic viruses which are also able to hide code; in fact polymorphic techniques are also adapted in some cryptors. In the case of runtime packers an arbitrary (and thus also malicious) program is taken, encrypted and furnished with a wrapper or stub which will decrypt the original program in memory just like a compressor extracts the payload as explained before. A cryptor can be seen as a weak form of a protector whose protection mechanism focuses on encryption to secure the payload-code; the high amount of available cryptors justifies its separate notation here. Besides many others, Morphine, Yoda's cryptor or PolyCrypt are some of the most popular encrypting runtime packers.

[7]UPX—the ultimate Packer for eXecutables: http://upx.sourceforge.net/

[8]ASPack: http://www.aspack.com/aspack.aspx

Protector: The third big category runtime packers can be grouped into are protectors. Whereas the compressors' ability to shrink executables also obfuscates them as a kind of secondary effect, the protector's focus undeniably lies in protecting its payload from reversing. This particularly reflects in the packed program's size; a compressor will try to keep the size as small as possible. A protector will strive to furnish the protected code with the most powerful defense mechanisms (cf. section 2.4) which, in virtually every case, leads to a bigger file size compared to the original file size.

This is why protectors are a good example of the shifted notion of the idea of 'runtime packers'. Output binaries of compressors are typically smaller that their original counterpart, whilst protectors produce bigger files due to added protection means. Still, both types add potency to the input binary. Some protection techniques (removal of symbolic information, some code obfuscation techniques, anti-disassembling) may not even require the packed executable to have a stub which is generally a typical characteristic of a packed binary. On this note, exponents from different categories of runtime packers can be distinguished from each other, normally not an easy task because purebred packers of one kind (like UPX as a compressor) are rather exceptional.

Many wide-spread protectors like Armadillo[9], ASProtect[10] or Obsidium[11] are commercially sold; there are also free versions like PESpin[12].

2.2.3 Malignancy of Runtime Packers

Unlike commonly-known suspects like viruses, worms or trojans, which are clearly linked to the realm of malicious software by the common computer user, runtime packers can neither be categorized definitely as malware nor as totally legitimate pieces of software; they remain in a gray area.

If someone inspects the methods of how a runtime packer does its work, at first glance it reminds of one of the techniques poly- or metamorphic malware takes advantage of. A cryptor encrypts program code to render it incomprehensible as well as a polymorphic virus does in order to hide itself from the fatal intervention of an antivirus software. A typical protector implements a conglomeration of anti-reversing techniques which are often connatural to those a polymorphic or metamorphic virus can resort to. While it is true that these methods may overlap, the runtime packers and classic malware vary in important matters.

The notion of a runtime packer's significance towards malignancy or benignity, respectively, has changed over time, along with its range of application. Originally designed for inoffensive usage as a way to save disk space, with time features were added and replaced to serve new purposes. Nowadays, stocked with this extended functionality, it is not only easier to use runtime packers for legitimate aims, but also easier to misuse them. Still, their *purpose* differs from a malware's intention.

In contrast to a protector, whose function is to safeguard some executable code, a virus'

[9]Armadillo: http://www.siliconrealms.com/software-passport-armadillo.html
[10]ASProtect: http://www.aspack.com/asprotect.aspx
[11]Obsidium: http://www.obsidium.de/show/home/en
[12]PESpin: http://pespin.w.interia.pl/

or worm's priority is to replicate and distribute itself (besides other malicious activities) rather than protecting its payload; the same applies for spyware or trojans. In these cases the obfuscation techniques that malware may be rigged with are only a means to an end in order to thwart antivirus software. In contrast to this, for obfuscators these techniques serve as their purpose. Runtime packers can support the protection of malware which makes packers dangerous. Neither in this case changes their purpose; they are merely misused.

Secondly, for most malware it is true that their obfuscation techniques only apply to their own code. A virus' obfuscation engine only hides *itself* but normally leaves the host's code alone. In comparison, a protector not only has to take care of its own code in an executable but also needs to protect the payload's code. As an objection, one could say that metamorphic code which is capable of code integration (as described in section 2.2.1) is an exception to this because here not only the virus' code but also the host's code is altered in every new generation of the virus. However, to overrule that, this technique merely serves to strengthen the obscurity of the virus' own code and not to guard the payload's code in some way.

In summary, on the one hand there is the semantic difference, the purpose that separates typical malware from runtime packers. On the other hand there is the technical disparity which manifests itself in the way that shared techniques are applied to different kinds of program code. These differences show that although obfuscation methods are used in both types of software, their application differs. Yet, the usage of runtime packers leaves a sour note as its application is not unproblematic in several cases.

2.3 An Insight into Executables

Before having a look at the functionality of packed binaries, it is reasonable to first gain some insight into executables themselves. It is important to know how a file, that contains the instructions which tell the processor what to do, looks like from the inside before learning how this structure can be altered and equipped with defensive mechanisms for protection. The way executables are designed differs from system to system. As this thesis is about malware, and malware is currently most common on Microsoft's Windows systems, we will concentrate on this platform when it comes to the specific implementation details of general concepts, described below.

2.3.1 Generation of an Executable

In order to start off doing something useful with computers, software is needed which tells the hardware what operations it needs to perform. Software is a very comprehensive generic term and one can think of hundreds of software units in different forms and shapes. A typical software package consists at least of one or mostly several, *binary executable files* which in their core contain machine code which is loaded, interpreted and finally executed by the processor. What seems to look like some arbitrarily arranged bytes to a person, for the processor constitute low-level instructions like basic arithmetical operations or moving data from one register to another. Elaborately arranged instructions like these are the basis for creating the virtual worlds of computer games, directing internet traffic or performing climate simulations.

Program Languages

Apart from the beginning of the use of electronic computers, people have never written machine code directly themselves but always made use of tools which actually create machine code for them. Today's complex applications could never be built without the use of high-level program languages which abstract the processor architecture beneath and enable developers to express their instructions in a form that more or less resembles human language or mathematical formulas. Furthermore, it allows one to give the program structure which in turn allows the programer to chop the application's complexity into many pieces which finally are supposed to work together like cogwheels in a clockwork. The tool which actually performs the translation from the high-level language to machine code is called a *compiler*.

Dependent on their closeness to machine-readable code, high-level languages can be grouped into several categories or levels [1]. Low-level languages of the first and second generation are the machine code and the assembly language: the former can directly be realized by the processor, the latter essentially is just a textual mnemonic substitute for machine code. Thus the programer can write easily recognizable commands like `MOV EAX, EDI` or `INC EBX` instead of the equal byte-pattern `8B C7` and `43`. An assembler translates this textual representation into binary code. Today's third-generation languages are commonly used to create sophisticated applications. Exponents like C, C++, Java or C# abstract the complex but fine-grained nature of assembly language and therefore make it easier for developers to express complex commands on the one hand but on the other hand also impair the amount of control the programer has over the subtleties of the computer's internals. Still, the advantages preponderate and the impairance of performance is compensated by the ongoing increase of processing speed of modern computers. Languages of the fourth generation increase the abstraction by specializing in certain applications, like querying databases (SQL) or formatting text (PostScript). The fifth generation allows programers to express logic and constraints by using languages like Prolog.

Language Processors

A *compiler* is a program which translates a program described by one language into an equal description expressed in another language usually of a lower level [1]. The output of a compiler is the program's representation in assembly language which in turn can be fed with a user's input and produces some output while executed by the processor. The compilation's process is complex and can be divided into several steps. In these steps the compiler analyses and checks the syntax and semantics before it builds an internal intermediate representation, which is used to perform some optimization in order to increase the efficiency and therefore the execution speed of the target code. Once the intermediate representation is optimized, it is translated into the target language.

In opposition to this concept are *interpreters* which skip the process of compilation and directly process the program-code of a higher language together with the user's input and produce some appropriate output. In comparison to a compiled program which is available in the processor's native code, an interpreted program is translated dynamically by an interpreter when executed. Interpretation normally leads to a significantly slower execution speed, however, the very time-consuming process of compilation is not necessary. Therefore, it is easier to develop more rapidly and, because the program's language is high level, it runs on any platform for which an interpreter exists.

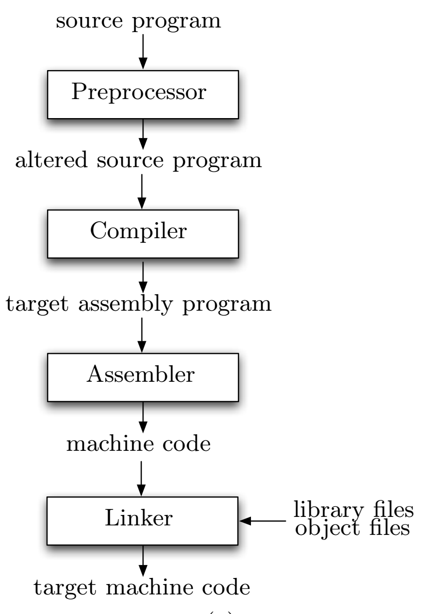
A *hybrid solution* compiles a program's source code into so-called bytecode, an intermediate form of the program. A virtual machine either interprets this bytecode when it is executed or utilizes a just-in-time compiler, which compiles the intermediate form into machine code dynamically every time and always just before the code's execution. The latter system is established in the Java language, for example. The benefit of such as system is that Java code in its intermediate form can be run on any platform as the virtual machine takes care of translating the bytecode into the particular platform's machine code. Optimization algorithms can be applied during compile-time to the intermediate form which again saves time whenever the bytecode is executed.

A *decompiler*'s job is to reverse the process of compiling and translating machine code back to a high-level language. However, it is a very challenging task to reconstruct the program in high quality, as generally much of the program's original layout gets lost during the compilation process. In particular the optimization process is prone to alter the code's organization as its goal is to reduce execution speed without considering the programer's original buildup which typically leads to a significant reduction of readability. A *disassembler* reverses the process of assembling, therefore it translates machine code into its mnemonic representation in assembly language.

Linkers, Loaders and Complementary Tools

A language-processing system which in its core consists of a compiler is normally supported by complementary tools which take over jobs to supplement the process of compilation: The *preprocessor* is used to make adjustments to a program before it is handed over to the compiler. On the other side of the chain once a compiler is done processing every module of a program a *linker* does the final job of putting all compiled pieces together, merging them into one executable and referencing all dynamically referenced libraries [20].

Dependent on its configuration, a linker can process different kinds of input and direct its output into files of various formats which in turn can serve again as input for a subsequent linking process. An *object file* is the most simple form of a compiler's output and represents a compiled program module. A given object file can arbitrarily reference other object code which is usually bound in library files, of which there are two forms: a *static* and a *dynamic library.* While both types are used to aggregate commonly used object code of a shared context in order to re-use it for different kinds of programs, the two libraries vary in the way they are applied. Whenever a linker links an object file with a static library, the required library's content is copied into the linker's output which,



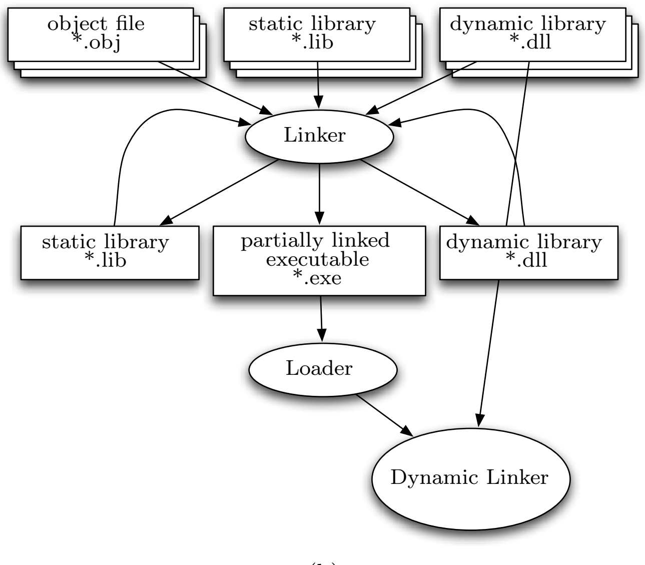


Figure 2.3: The diagram on the left (a) shows the flowchart of how a program written in a high-level language is transformed into equal executable machine code. A compiler is fed with the pre-processed source program, while the assembler translates the compiler's output into machine code. The linker (b) is responsible for putting all assembled parts together in order to produce a file that the operating system can execute. Optionally a linker can also produce a static or dynamic library which in turn can be re-used by a subsequent linking process to get another output. Both diagrams are taken and adapted from [1, page 4] and [17] respectively.

for example, is an executable file. Thus, afterwards it is not possible to determine which code of the final executable came from the object file and which one was taken from the static library as their contents were merged with each other. In contrast to this, the code of a dynamic library is only recorded as a reference in the output executable.

The advantage of this approach is that it adds flexibility to the program's structure as parts can easily be replaced or updated, instead of having to deal with a big monolithic package that has to be replaced as a whole each time even a small change has been made to one of its modules. For example a new and better implementation of an algorithm can be used by a program by just replacing the appropriate library file. Secondly, it saves space on disk as well as in memory because different processes share code in one external library and do not require a copy of this code in each of their instances. The downside of this convenience is the increased complexity of this system.

Hardly surprisingly a program *loader* is responsible for loading a program into memory and preparing it for execution. This process is not as trivial as one might think because address *relocation* has to be taken into account.

Virtual Memory and Relocation

The parallel execution of multiple processes is something very common today. While one or more instances of a browser allow the user to surf the web, a media player takes care of diversion and entertainment and the operating system performs some maintenance job in the background. It is important not to let all these processes get in each other's way and make the system crash. The *MMU*[13] of the CPU is responsible for *memory protection* and *relocation* among other things that are not of so much interest here (cf. [40] for thorough information).

An important concept related to this context is *virtual memory*. While in the process' view it has one big empty memory block of contiguous memory addresses at its disposal, the memory pages actually allocated and addressed by the process are only virtual. This means the memory addresses that the process is aware of, occupies and accesses don't correspond to those in physical memory that are actually allocated. To make it possible to let several processes simultaneously reside in memory without taking into account this very requirement during compile-time, the MMU acts as a level of indirection and may place the process' code into free physical memory, split the supposed contiguous code block into fragments or could even swap it to secondary memory. It takes care of mapping every virtual memory address with its counterpart in physical memory. This layer of indirection relieves the program itself from being responsible for memory management issues and allows the operation system to execute processes with flexibility.

Memory protection ensures that it is not possible for a process to erroneously access memory areas of another process and therefore cause the system to become unstable. As a process has to use system functions to obtain allocated space in memory for its own use, the operating system is able to keep track of these memory pages for every process and prevent it from illegitimately accessing other addresses that a concurrent process may use. The technique of virtual memory allows the memory areas of one single process to be fragmented and arbitrarily scattered over the computer's physical memory. Infrequently parts of processes may even be transparently swapped to secondary memory in order to make place for other processes, thus allowing them to use fast and therefore more expensive and scarce RAM.

According to [20] a linker's merging process also includes link-time *relocation*. This means that in Windows, the linker determines where in memory which modules of the program should be loaded and updates the references to global memory addresses directly in the machine code. As this concerns virtual addressing, two instances of one program do not know of each other's existence and address the same memory locations but the MMU maps them to different physical addresses and therefore allows two instances of the same program to run simultaneously in memory. The linker's pre-determination as to where in memory the program should be run significantly speeds up the process of loading an executable because the loader doesn't need to care about updating any references in the program's code anymore but only needs to load the executable into memory where the linker intended it to be and it is ready to run.

As far as statically linked modules go, there are no problems with this linking/loading

[13]**M**emory **M**anagement **U**nit

strategy, however, this is different whenever dynamic libraries need to be loaded as well. When an executable needs dynamic libraries for execution, the loader will realize this and load the required libraries into the process' memory as well.[14] A dynamic library in Windows (DLL[15]) also has a preferred location where the linker intends it to run. So, it can happen that one library is required which per default uses the same address space that is already occupied by another dynamic library which has already been loaded before. In this case it is the loader's responsibility to choose a different base address to load the second library to. Once relocated, all references to global memory addresses point to wrong places in memory, meaning the linker also needs to add the relocation offset to all relevant references; the same work the linker has already been doing for the default base address. These references are recorded in a special place within the DLL for this purpose. Because the relocation process is very time-consuming, it is generally advisable to coordinate the base addresses of dynamic libraries in a software package and configure the linker to prepare the DLLs to use non-overlapping address spaces.

2.3.2 An Executable's Internal Layout—PE

The output of a compiler is not just plain machine code. Modern executable files follow an internal structure which meet the need for flexibility and portability in execution and help to overcome the consequential complexity of state-of-the-art computer systems.

According to [20] in the Unix world, a.out was a simple executable file format which already consisted of a header and several sections into which executable code was partitioned. With the advent of System V[16], a.out was later superseded by COFF. In turn COFF was again replaced by ELF in later versions of System V, still used in contemporary Unix systems like Linux and diverse BSD flavors like OpenBSD or FreeBSD.

In the MS-DOS world, at first there existed the COM format which actually did not contain any internal structure but was just pure machine code, replaced by the MS-DOS EXE format in later versions of MS-DOS. Both of those 16-bit formats were superseded by the advent of the first NT operating system, Windows NT 3.1. For executables running on 32-bit platforms like NT and also Windows 95/98/ME there was the new *PE file format* which, since then, is still in use even on Microsoft's latest NT incarnation, Windows 7. PE is the format we will concentrate on as most malware misuses this format to execute their malicious payload. PE is an abbreviation for Portable Execution and reflects Microsoft's intention to make PE a format which can be used by several implementations of NT on different platforms. Certainly the compiled program code would differ for each processor but the goal is that development tools don't need to be newly implemented for every future platform Microsoft is going to support [29]. In Windows systems PE is used for executable files (.exe), for dynamic libraries (.dll) and others (.drv, .sys, etc.) which all

[14] One has to distinguish between a local dynamic library which will be loaded for every instance of a program in its own private memory space and a global dynamic library which typically is provided by the operating system and will just be referenced by a new instance. Here we consider local ones.

[15] **D**ynamic **L**ink **L**ibrary

[16] System V was AT&T's commercial version of Unix released in the mid-80's. Read [40] for more on this topic.

feature the same layout but can distinguished from each other by a flag that is set in the header.

At its heart PE is the COFF format (the same used in UNIX systems before ELF) with extensions. This reflects in the glossary of the PE's specification [25] but also in the use of RVAs[17]. An RVA is used as an offset within an executable in memory. As mentioned before, an executable is loaded into its own virtual memory space with a specific predetermined base address. In the header there are pointers to special locations within the PE file. These locations are for example the beginning of a certain section or the entry point where the very first instruction that the processor will fetch is located. Those pointers are all relative to the base address that the loader put the file into its virtual address space. If, for example, in the PE header the address of the entry point is denoted as `1280` and the base address is `400000`, the actual entry point in memory will be at `401280` within the `.text` section. The purpose of the RVAs' usage in the header is to make the pointers independent from the base address that the loader puts the executable into. For those few references in the PE header (cf. *relocation* in section 2.3.1) the loader just adds the base address to the RVA in order to reach the designated location.

The internal organization of a PE file consists of the PE header, which is actually a conglomeration of several headers from different standards and the machine code with its metadata which is divided into semantically different sections (see figure 2.4). We shall only discuss the most important issues here; for more detailed and complete information about the PF file structure refer to Microsoft's PE specification [25] or Levine's book [20].

MS-DOS stub

In order to maintain backwards compatibility a PE file actually starts with the signature of an MS-DOS EXE file consisting of two bytes `4D 5A`, thus `MZ` in ASCII Code which stands for Mark Zbikowski, the developer of the EXE format. The magic number is followed by an actual runnable DOS program just printing an informatory text like "This program cannot be run in MS-DOS mode.". This way, if a user tries to run an 32-bit .exe file in a legacy MS-DOS system, he at least gets informed about the incompatibility. In the case of a PE-loader launching this executable it just ignores the MS-DOS stub and jumps to the PE signature, whose location is referenced by a pointer at a fixed address `3C`.

PE signature

The PE signature precedes the COFF header and consists of four bytes `50 45 00 00` which resembles the ASCII string `PE\0\0`.

COFF header

The COFF header holds a few important pieces of meta information about the PE file. The kind of platform this executable is intended to run on is stored here for example.

[17]**R**elative **V**irtual **A**ddress

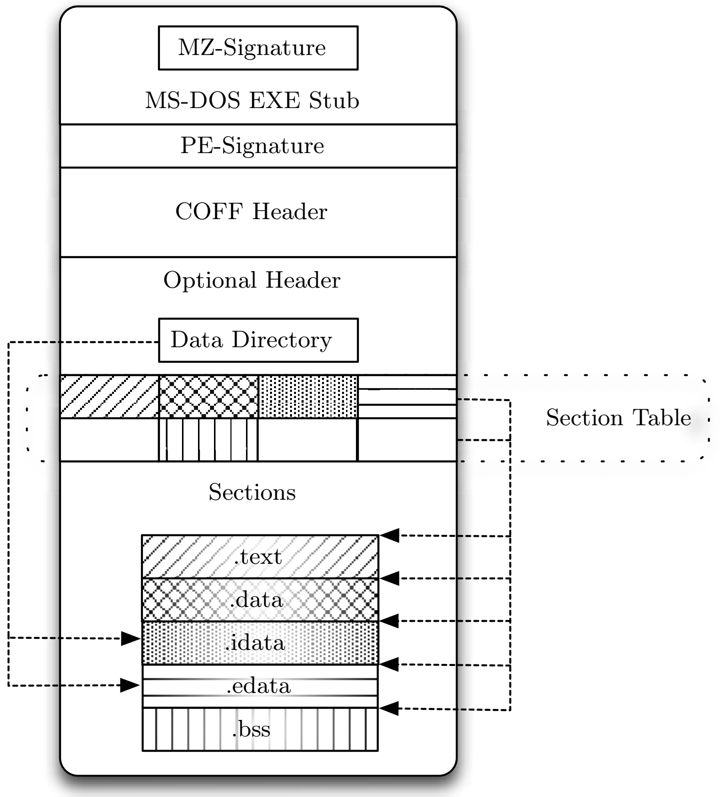


Figure 2.4: The internal structure of a PE file.

Furthermore the number of sections, the size of the following 'optional header' and also a flag that determines if the object file is an executable or a dynamic library.

Optional header

The optional header is an extension of the COFF header and—despite its name—is actually not optional. It retains some more necessary meta information about the executable itself and pointers to significant locations such as RVAs. The entry point is stored in the optional header as well as the image base (the address where the file begins in memory and the value which is added to RVAs in order to get the real address) and the size of the following section table. The *data directory* at the end consists of several RVAs to important data structures. An example of this is the import table which acts as an auxiliary level of indirection in order to store the addresses of dynamic libraries. See below for more information.

Section table

The section table immediately follows the optional header; this is important as there is no direct pointer to it but it is calculated using the preceding headers. Essentially it is an array of headers for the sections whose size is determined in the COFF header. Besides the entries for the section's name, and its size and two pointers (one to the beginning of the section in the file and the other one pointing to the virtual address where it will be mapped in memory), there is another pointer to the section's own relocation information. Furthermore, there are flags which determine the characteristics of the referenced section. The flag can indicate a section containing executable code or data which is either initialized or uninitialized.

The sections

The actual machine code is divided into sections which typically have varying properties and are treated differently dependent on the characteristics defined in the appropriate section header in the section table. The name of the section usually denotes its role and though not mandatory, follow a naming convention which makes it easier for a person to determine what a certain section is there for. The most important ones are the following:

- **.text:** the actual executable opcode.
- **.bss:** uninitialized data; i. e. global static variables whose values cannot be determined by the compiler during compile-time.
- **.data:** the same as .bss but the variables are initialized.
- **.edata:** Export Table; A list of symbols which are visible and can be used by other modules. An export table is typically used by a dynamic library to declare all functions and variables which can be used by executables.
- **.idata:** Import Table; The complement to the export table which indicates what external symbols need to be referenced. Here the loader looks up the required dynamic libraries and their symbols, which need to be loaded in order to be able to run the executable which imports those symbols.
- **.reloc:** relocation information; The area where the linker stores the information where the pointers to global memory locations are, which need to be updated whenever an executable needs to be relocated.

Import Table

An important data structure is the *import table* which is responsible for finding and referencing dynamic libraries. Protectors, cf. section 2.2.2, often target this structure in order to prevent reversing. It is probably not a mistake to have a closer look at this data structure.

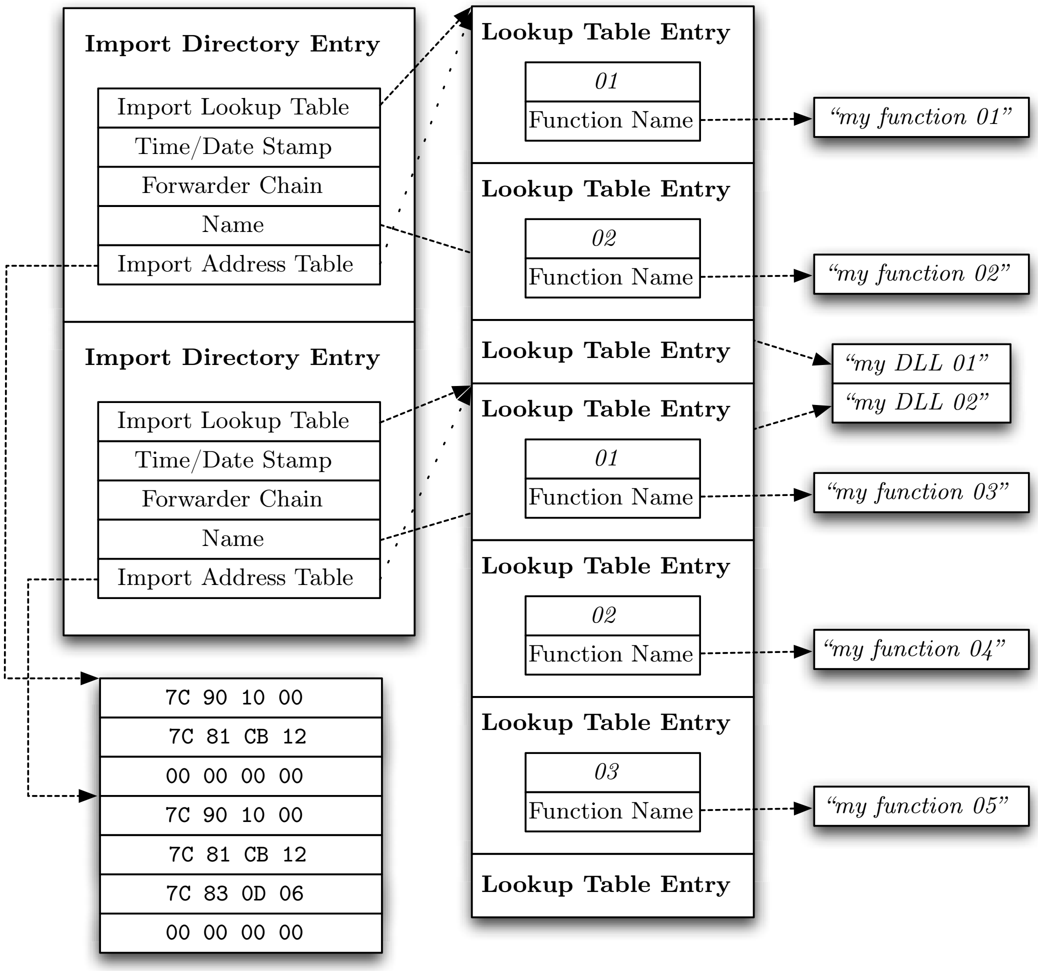


Figure 2.5: The layout of an import table.

The Import Table is located in the .idata section and pointed to by the second entry in the data directory. It begins with an array of Import Directory Entries, each of which represents one important dynamic library and, as a structure has got three important fields amongst others. The end of the Import Directory is denoted by an empty entry.

The first entry points to the Import Lookup Table, which again is an array of entries whereupon each one represents a symbol of the DLL pointing at it. Every function or global variable of a DLL can be referenced by an ordinal or its name. Thus every entry of the Import Lookup Table holds the accordant ordinal or a pointer a ASCII string that resembles the symbol's name.

An Import Directory Entry also holds a pointer to the DLL's name and as the last entry, a second reference to the Import Lookup Table. Later, once the DLL is readily loaded into memory, this second entry is overwritten by a pointer which references the actual addresses of the DLL's functions. Akin to the Import Lookup Table the *IAT*[18] is an array of addresses ordered by the function's ordinals and terminated by en empty entry.

This system solves the problem that during link-time, the linker has no knowledge of where a specific function of an external DLL will be during run-time. So, the linker does not know what address it should specify as an argument to the jump instruction, which in turn will direct the program flow to the external function of the dynamic library. The dynamic linker could update all references to the function when the executable is loaded, however, this is a brute-force method and too time-consuming. Instead the linker leads all jump instructions to a specific DLL's function dereferencing the address where the actual function's address will be put in the Import by the dynamic linker. Once the dynamic linker has constructed the IAT and updated the references to it in each Import Directory Entry, the program is ready to fully access the DLL.

2.4 Runtime Packers: Functionality and Obfuscation Techniques

Though we have seen that there are different types of packers (cf. section 2.2.2), the basic technique of how a packer alters an arbitrary executable stays more or less the same for all of them. While sometimes the mere process of ciphering or compressing is enough in terms of a cryptor or compressor, it won't suffice for protectors whose main goal is to complicate the reversing of the executable. Additional technologies extend the simple packing process in order to enhance the side-effect of obfuscation when an executable is packed.

2.4.1 Basic Packing

A packer acts like a wrapper around the original executable, called the *payload*. The packer may alter the payload at its discretion, as long as it is able to restore its original state to the degree so that it can be executed and will behave in the same way as if it was not altered at all. As described in [43], when a packer is instructed to pack a PE file, it scans the internals and re-structures the headers along with the sections and may also alter the import and export tables. Dependent on the implementation of the packer it takes the executable as a whole or just takes important sections for code and data and puts it into a single custom section which will be encrypted or packed. In order to be able to transparently restore the executable when it is executed, the packer will also attach a new code segment called a *stub* which takes care of this issue.

When a user opens a packed file for execution the loader will set the IP[19] to the address specified as the EP[20] in the Optional Header. The EP normally points to the

[18]**I**mport **A**ddress **T**able

[19]**I**nstruction **P**ointer: a register that holds the address of the instruction which is going to be executed next.

[20]**E**ntry **P**oint: pointer in the PE header indicating the first instruction of an executable.

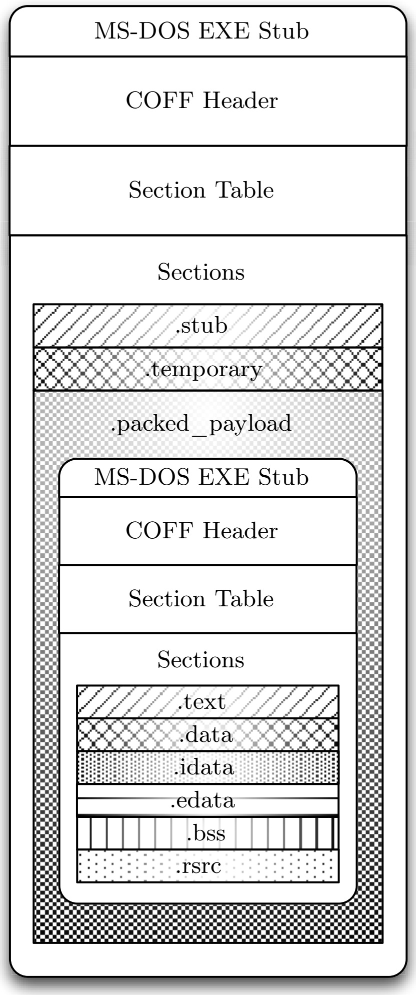


Figure 2.6: This simplified diagram shows a packed PE file. The packed original executable is wrapped as payload inside a custom section that is part of the wrapping PE file created by the packer. When it is executed the stub-code residing in a separate section (in this case called '.stub') takes care of unpacking the payload, prepares it for execution and finally hands over control to the packed PE file.

first instruction in the opcode section of an executable, however, in this case it was changed by the packer and now points to the beginning of the stub. Primarily the stub saves the register context so that it is later able to restore the exact state in which the original executable was supposed to be run. Typically this can be archived by a `PUSHA` instruction, which pushes all common registers onto the stack. Afterwards it starts with the process of unpacking or decrypting the payload into a pre-allocated buffer in memory. Once this is done, the stub needs to load and link the dynamic libraries which the original executable requires to do its work.

As already discussed earlier, this job is normally done by the loader, however, at loading time the packed executable's import table was still hidden and not in its place. The stub has to do the loader's work manually, as the loader took only care of the stub's dependencies which are identifiable from outside by it. Last but not least a `POPA` instruction reverses the `PUSHA` instruction, restores all relevant registers and therefore reconstitutes the context as it was before the stub began its job. Finally, the control is passed to the packed program as it is now ready to run. This is usually accomplished by a `JMP` instruction to the accordant address where the OEP[21] of the packed executable is located. If the reconstruction of the payload was successful, now the process continues with the original task it was supposed to perform.

This is how a basic packer or cryptor works. The payload gets encryped or packed and is unwrapped automatically and transparently by the stub in memory, continues with its original task afterwards and saves disk space assuming that the wrapping code is smaller than the amount of space saved by the compression algorithm. However, in order to prevent or at least impede reversing these means is not enough. With some expertise it is quite easy to manually unpack and recover the packed executable, even though automatic unpacking is possible. Basic packing only complicates reversing to a small degree. Protectors make use of additional techniques that may drastically extend the anti-reversing abilities in comparison to a traditional packer. A variety of these techniques are presented in [12] and [16] and can be categorized into several groups:

- **Eliminating Symbolic Information:** Though during the compilation process the fine-grained assembly code loses much of the readability of its equivalent high-level code, experienced reversers can find many clues which help to find out about the internals of an executable. Languages which compile into an intermediate bytecode especially contain lots of textual symbolic information about class names, their fields or global objects. With annotations like these it is enormously easy for a reverser to do his work. Also executables that contain native machine code include symbolic information. Packers usually remove them if not essential or hide them if they are crucial and the file cannot be run without it. In the case of verbose intermediate bytecode of languages like Java or .NET all meaningful symbols should be renamed to random strings that will not provide any internal information for reversers anymore.

- **Code Obfuscation:** When someone talks about obfuscation of program code he means one of several techniques that aim to complicate static analysis. Reversers can orientate themselves to the executable structure, so that obfuscation changes "the program's layout, logic, data, and organization in a way that keeps it functionally identical yet far less readable", as Eilam puts it in [12, page 329]. Obfuscation can be applied in many ways and there are numerous techniques for it.

- **Anti-Debugging/Anti-Disassembling:** One of the most important tools a reverser usually uses is a debugger. It makes it possible to dynamically analyze a pro-

[21]**O**riginal **E**ntry **P**oint: the EP of the original executable before is became packed.

cess and to step through its instructions in order to find out what task a process is currently performing. A packer may insert anti-debugger code that will detect the presence of a debugger attached to the process and react on this situation. This can mean that the code will passively impede the disassembly which results in meaningless garbage assembly code or it detects the debugger during execution and takes active measures such as terminating itself or other, more sophisticated interaction with the debugger. These active defense mechanisms are often debugger specific as they exploit individual peculiarities of certain debugger implementations.

- **Antidumping/Anti-interception:** Dumping is a well-known technique to restore a packed executable. It takes advantage of the fact that an executable's layout in memory does not much differ from the same file on disk. By halting the process of a packed file at the right time—which generally means at the moment when the stub has done its work, when the process is just about to jump to the OEP and the executable's code is unpacked in memory—it is possible to write the program code in memory to disk, perhaps make a few adjustments and the program is unwrapped again. Antidumping and anti-interception are some of the techniques to prevent either dumping itself or to make it difficult to find the right instruction for where to halt the process and dump it to disk.

- **Anti-Emulation:** One approach to find out if a packed executable includes malware is to let it run inside an emulator and see if malicious code or fragments unravels; an emulator prevents malware from doing any harm. Anti-emulation code tries to realize its environment and changes its behavior and will not execute any malicious routines or terminate if it comes to the conclusion that it is probably being analyzed.

- **Emulation-based Obfuscation:** The latest generation of packers aiming to foil reversing also use the technique of emulation which gives headaches not only to antivirus researchers. Here, arbitrary parts of the original executable are translated into a custom machine language with a distinct instruction set which is uniquely created for every process of packing transformation. It is understood that this bytecode is not able to run on platforms that the original executable was intended to run on, so the wrapper provides each packed executable with an individually adapted stub that takes care of dynamically translating each instruction of the custom opcode into native machine code. The unpleasantness from the perspective of those who want to break the system is that, in opposition to previous techniques, there was always one point in time when the whole packed code was in plain and ready to run (and dump) in memory. This is no longer the case as instruction by instruction get translated, is emulated and is never stored permanently into memory. In combination with other potent techniques emulated executables become very hard to be reverse-engineered.

The next sections describes anti-reversing techniques in detail.

2.4.2 Eliminating Symbolic Information

As said before, some useful information in the form of text strings find their way into compiled executables which can be analyzed by reversers to gain internal information about it. As for PE files that are compiled into machine code information of this kind is only embedded when they are compiled for debugging which is normally not the case for programs that are distributed and sold. In the sections for import and export data there are still textual representations of the linked dynamic libraries and their functions. This makes it easy to draw inferences on the functionality of that piece of code that makes use of these external functions. Developers who want to eliminate these clues in their assemblies should too consider the use of ordinals for linking with external libraries, instead of letting the linker write the API's names into the PE file.

Modern languages that compile source code into intermediate bytecode implement sophisticated techniques (e. g. reflection in Java) that require names for internal cross-referencing and embed this information into the compilation. This makes it possible to decompile it into a very well-readable program in high-level language. As the cross-referenced names cannot just be stripped from the program without breaking its functionality, the strings can be renamed to meaningless text without breaking the references. Obfuscaters can perform this task automatically. They take compiled programs in bytecode as input and provide fully functional but obfuscated programs as output.

2.4.3 Code Obfuscation

When someone performs reverse engineering he or she always tries to gain information from a compiled program which is actually not meant to be read and understood by humans. Still, given enough expertise and time there is a very high possibility that one is finally able to find the information he or she is looking for. Obfuscation aims to add useless and meaningless code to the program for the purpose of hiding relevant information within. This way a potential reverser cannot distinguish between useless and significant code anymore thus rendering it very hard or even impossible to understand how the program works and what it actually does. While in some cases obfuscation bloats the code, which may lead into slower execution speed, increases the need for memory and for disk space, this is not true for some techniques which just re-organize the program's structure and will not impede the execution of the program but still confuse a human reverser.

Opaque Predicates

Opaque Predicates described in [10] and [12] are a smart form of obfuscation that aim to confuse both humans and deobfuscators. Here additional condition statements are introduced which check tautologies or contradictions. A tautology is a condition that is always true, but a contradiction is its opposite and will always be false. An example of an opaque predicate using a tautology is the following:

```
if (foo != foo + 1) {
  /* branch is always taken */
} else {
   /* branch is never reached; place useless and confusing junk code here */
}
```

As the condition can never become false the first branch is always taken, while the second one will never be used and can contain deceptive code for confusion. Of course it is not a big deal to realize that this very code example looks somewhat suspicious. One can think of more complex structures that firstly blend into the normal program flow and secondly whose behavior is hard to predict for humans without actually letting the program run and see where the program flow goes. As to deobfuscators, the condition should be so complex so that neither they can deduce its outcome. An example presented in [10] takes advantage of aliases to construct opaque predicates. Aliases are pointers that have different names, but point to the same object in memory. It is quite difficult for deobfuscators to resolve aliases; they have to keep track of which pointer points to which object at any given time.

The structure in this example consists of several objects, called nodes, which are segregated into two or more groups, G and H in this case. Additionally there exist some global pointers which act as aliases for the nodes in G and H, called g, h and f. The nodes are partially interconnected within one group and each node has a boolean field that can be set to either true or false. The obfuscator introduces bogus statements which casually change this structure, for example by adding or removing nodes, moving nodes to another group or merging two groups. However while doing this, some secret invariants, unobvious rules are kept that never change. For example "the pointers g and f never point to two nodes that are kept in the same group", "there is never a connection between those nodes pointed to by f and h" or "the flags of those nodes pointed to by h and g will always differ". The structure set up in this example is complex enough to think of many of those rules. Simple conditional statements are then introduced which check properties of this structure. Some of those statements can turn to either true or false dependent on how the structure was changed before by the bogus instructions. A few others, however—those which check the invariant rules that are never broken by the bogus statements—always yield the same output. These are the opaque predicates in this system. For a more detailed explanation see [10, page 5f].

The benefit of this system is that it is stealthy because assigning objects to pointers is not uncommon in languages like C, C++ or Java. It is resilient because 'destructive update' operations i. e. overwriting a memory address of a pointer with another one, making it point to a different object, is very hard to process for deobfuscators. Thirdly, the creation of a system like this and handling it is not difficult, therefore making it cheap.

Table Interpretation and Ordering Transformations

Table interpretation [12] is the term for a simple technique that has a high degree of potency regarding both humans and deobfuscators. Here a particular set of instructions, preferably a function, is split into several pieces which are mixed together. Furthermore there exists some control code together with a jump table which directs the program flow to the next right set of instructions. Whenever a certain code segment is finished, it sets a specific register whereupon the control code uses this information to go to the right index in the jump table and to jump to the address of the following code segment. As the split code breaks all cohesive structures like loops or conditions and their branches, the degree of confusion is high because one has to arduously recollect the pieces and put them together in the right order to understand their purpose. This effect can be aggravated by combining this technique with other anti-reversing techniques like anti-debugging code.

A technique that is akin to table interpretation is *ordering transformations*. Here a set of instructions is again analyzed and then those instructions are randomly reordered. Of course this cannot be done arbitrarily but for reordering, instructions need to be chosen that are not dependent on each other; i. e. whenever an instruction y requires another instruction x to be executed before, they cannot be interchanged. The confusion stems from the fact that a human expects code to be semantically ordered.

In- and Outlining

Inlining is a code transformation technique used by optimization algorithms of compilers. Instead of implementing a function call actually as a function call, it copies the whole function's body to the place where a call to this function should be. This saves the overhead of creating metadata for the function on the stack as the callee's code is embedded in the caller's code. But certainly space is wasted as the same code is duplicated to several places within the program. Outlining on the other hand means to take some arbitrary code, copy it to somewhere else in the program's memory space and replace the original code with a function call to the outsourced code.

A combination of in- and outlining may help to reduce the readability of program code [9]. In order to increase potency and resilience it is suggested to inline some function F whose code is then merged with the caller's instructions C, dividing C into two parts C_1 and C_2. In doing this several times some obfuscation is already applied to the code. This effect can even be enhanced by outlining some chunks of this newly generated code again. For this purpose the outlined code should contain instructions of both the caller and the callee. As there exist two intersections between caller's code C and the inlined function F (once at the beginning of F's code where the code of C_1 passes into the functions code and a second time at the end of F when the caller's code C_2 goes on again) the code for outlining is taken from the area of those two intersections. This way the two new outlined functions contain code from the end of C_1 together with the beginning of F and code from the end of F and the beginning of C_2. As described before the code of those outlined functions is replaced by function calls. By applying this technique several times, pieces of code become more and more intertwined and distorted.

Interleaving Code

Yet another method to obfuscate the program flow is to interleave the program code. Program 2.4 shows how this is done. Two or more functions are taken and parts of their bodies are mixed together. At the end of every part there is a jump instruction to the

```
Function1() {
  Function1_Segment1;
  Function1_Segment2;
  Function1_Segment3;
}
Function2() {
  Function2_Segment1;
  Function2_Segment2;
}
Function3() {
  Function3_Segment1;
  Function3_Segment2;
}
```

```
Function1_Segment3; f1_end();    // End of Function1
Function1_Segment1;              // This is the Function1 entry-point
Opaque Predicate                 // Always jumps to Function1_Segment2
Function3_Segment2; f3_end();    // End of Function3
Function3_Segment1;              // This is the Function3 entry-point
Opaque Predicate                 // Always jumps to Function3_Segment2
Function2_Segment2; f2_end();    // End of Function2
Function1_Segment2;
Opaque Predicate                 // Always jumps to Function1_Segment3
Function2_Segment1;              // This is the Function2 entry-point
Opaque Predicate                 // Always jumps to Function2_Segment2
```

Program 2.4: All functions segments are taken and mixed together. To maintain the original program flow opaque predicates are use to implement jumps to the right statement. Code taken and modified from [12, page 354].

next part, meaning the instructions are disordered while the program flow is not changed. The jump can be implemented as *opaque predicate* as the whole structure can be quickly unscrambled by a deobfuscator if unconditional jumps are used. To make it perfect all interleaved functions may share the same entry point but an additional parameter decides which function is actually executed, especially confusing if all the functions don't share any semantical commonness [12].

Parallel Code

Code may not just be spilt and intertwined but can also be distributed over two or more parallel flows of control [9]. Forking one linear flow into multiple others is no problem if the instructions don't have any data dependencies. For those which do, the application of semaphores helps to avoid race conditions. By creating several threads or processes which share the execution of the main program potency is increased as an obfuscated parallel program is more difficult to reverse than one with sequential control flow.

Data Transformations

The next technique doesn't aim at obfuscating the program flow but the program's data. Eldad Eilam suggests the following methods to do so in [12]. One is to change the variable's encoding which means to deviate from the 'natural' way to store data.

If a programer wants a loop to count from 1 to 50 this is how he would normally implement this:

```
for (int i = 1; i < 51; i++)
  out(i);
```

But the following code does the same:

```
for (int i = 6159; i >= 132; i =- 123)
  out(51 - ((i-9)/123));
```

Obviously the second piece of code is quite more complicated than the first one in 'natural' notation.

Another possibility is to avoid constants but to create a function which yields the needed values. One could think of a function taking an integer value as input and returning a text string dependent on the input. Furthermore the function is obfuscated by loops and conditional branches that react differently depending on the input and compose various strings as output. The correct behavior is only triggered by supplying the appropriate 'secret' input value.

A third example suggests merging two variables into one. This can be done on bit-level; so, two variables x and y, both 32 bit long can be merged into one 64 bit variable g, whereas the least significant bits of g may represent x and the most significant bits represent y. Further obfuscated by applying an operation with a random value r, the result may scramble the two 32 bit variables well.

Polymorphism

The typical packer's design is very inviting to be extended by polymorphism. As described in section 2.2.1, a polymorphic virus uses a stub to decrypt its body before control is finally passed on to the decrypted payload which then performs some kind of operation. Other files are finally infected with another version of the virus that is encrypted with a different key, so that the payload of the new virus' version differs from the original byte-code. This method and more refined techniques make it difficult for anti-virus software to pin down a specific virus by traditional means, for example the signature-based approach. In order to protect the payload a polymorphic runtime-packer may adapt polymorphic or even metamorphic techniques easily to increase the quality of the obfuscation applied to the system that is formed by the packed binary and the decryption stub.

2.4.4 Anti-Debugging/Anti-Disassembling

Instead of trying to make the code itself confusing and incomprehensible anti-debugging aims at thwarting the debugger, one of the reverser's most important tools. One can distinguish between active and passive measures. Active measures try to find evidence that the process is controlled by a debugger and reacts to this perception by altering the program flow. This can mean that the process will terminate itself but it may also attack the debugger itself by exploiting peculiarities or vulnerabilities. In contrast to this, passive attacks try to confuse disassemblers by introducing incorrect opcode which leads to incorrect interpretation of the following opcode, that is actually valid. The following attacks are originally described in [12] and [16]:

Anti-Disassembling/Passive Attacks

A disassembler parses the opcode of an executable to display it in comprehensible mnemonic form.

```
--------   55           PUSH EBP      ;Push the value in reg. EBP onto the stack
--------   89E5         MOV EBP,ESP   ;Copy the value in reg. ESP to register EBP
--------   E9 6CFFFFFF  JMP 0040104E  ;Jump to address 40104E
```

While contemplating these examples, it is easy to realize that the instruction set for Intel's widely spread x86 processor architecture constitutes of instructions of variable length. This fact is exploited when a packer inserts anti-debugging code, yet the debugger's parsing algorithm is crucial to the success of the attack.

Disassemblers that use a technique called *linear sweep* for disassembling are only fed with the stream of bytes that constitute the program's executable code. Instruction by instruction is decoded while the debugger is not aware of any structure in the code. Normally this goes well as long as the disassembler doesn't stumble over faulty code. The trick is to put some opcode at an appropriate place that looks like the beginning of an instruction but is actually only the first part while the second part is missing. However, the disassembler interprets the following bytes (which are actually part of the next valid instruction) as the appendant and desynchronizes that way. Desynchronization means that due to the wrong interpretation of the previous bytecode the disassembler also fails to interpret the following instructions correctly, as it begins to decode any new instruction at the wrong place. It mistakes a certain byte, which may actually be the middle part of the next instruction, for the beginning of this instruction. As for the x86 instruction set, the disassembler resynchronizes agains but at least for a few instructions the damage is done. These flawed opcode bytes are typically placed where the normal program flow would never reach them and cause the CPU to misinterpret them. For example a jump instruction may direct the flow over the the faulty bytes to the next valid instruction. As said before the linear sweep disassembler parses the whole code and therefore will trip while the processor never notices that something is wrong.

OllyDebugger along with other modern disassemblers, use *recursive traversal* for parsing the program code. This more robust technique follows control flow instructions to disassemble only those bytes to get a better clue of which bytes are actually legitimate. Unfortunately it is still possible to lure the disassembler into picking up junk bytes by deploying *opaque predicates* (cf. section 2.4.3). During runtime it will always direct the program flow to the executable branch. If the opaque predicate is constructed cleverly enough it can fool a disassembler using recursive traversal into parsing also the faulty branch which will generate the previously described effect again.

The assembly code below taken from [12, p. 337] serves as an example to demonstrate how various disassemblers react.

```
  jmp After
  .byte 0x0f
After:
  mov eax, [0x00400010]
  push eax
  call eax
```

This is the output OllyDbg[22] gives when fed with the compiled executable:

```
00401378        EB01            JMP SHORT antidbg1.0040137B
0040137A        0F              DB 0F
0040137B        A1 10004000     MOV EAX,DWORD PTR DS:[400010]
0040137D        50              PUSH EAX
0040137E        FFD0            CALL EAX
```

Here is the output of WinDbg[23]:

```
00401378 eb01           jmp     image00400000+0x137b (0040137b)
0040137a 0fa1           pop     fs
0040137c 1000           adc     byte ptr [eax],al
0040137e 40             inc     eax
0040137f 0050ff         add     byte ptr [eax-1],dl
00401382 d0c7           rol     bh,1
```

Obviously OllyDbg has realized that the abnormal byte is no meaningful executable code and has declared it as data. WinDbg on the other hand has stepped into the trap and shows only nonsense code after the `0F` byte. It interpreted the junk byte as actual executable code which results in a `POP` instruction consuming another byte `A1` which was actually supposed to be the beginning of the legitimate `MOV` instruction. When we step through the code past the jump instruction WinDbg realizes its mistake, shows the right disassembly henceforward, cuts the prior disassembled code and displays a somewhat puzzled message “No prior disassembly possible.”.

Similar code extended with opaque predicates is also able to mislead reverse traversal-based debuggers. In the following code (example taken from [12]) the conditional jump is never taken as `0x23` will always differ from `0xff`. The subsequent code shows OllyDbg’s output.

```
  mov eax,0x23
  cmp eax,0xff
  je Trap
  jmp After
Trap:
  .byte 0x0f
After:
  mov eax, [0x00400010]
  push eax
  call eax
```

[22]http://www.ollydbg.de/

[23]http://www.microsoft.com/whdc/Devtools/Debugging/default.mspx

```
0040135B      B8 23000000       MOV EAX,23
00401360      3D FF000000       CMP EAX,0FF
00401365      74 02             JE SHORT antidbg2.00401369
00401367      EB 01             JMP SHORT antidbg2.0040136A
00401369      0FA1              POP FS
0040136B      1000              ADC BYTE PTR DS:[EAX],AL
0040136D      40                INC EAX
0040136E      0050 FF           ADD BYTE PTR DS:[EAX-1],DL
00401371      D0C7              ROL BH,1
```

One can see that in this example OllyDbg also fails at disassembling the primed code. Apparently it is not able to realize the opaque predicate and that the jump to ‘Trap’ is actually never taken. This misleads OllyDbg into parsing the invalid junk byte which leads into the same invalid junk code we have already seen before.

Anti-Debugging/Active Attacks

Active attacks search for traces of a debugger and take accordant measures which, for example, can be the termination of the debugged process. There are numerous ways to detect debuggers; active anti-debugger code is particularly platform- and debugger-dependent, so there are hardly any silver bullets to hit all technologies. In [16] Peter Ferrie presents a good deal of elaborate active anti-debugger techniques, some of which can be embraced into groups, while others are a loose set of ingenious tricks of how to elicit the process environment information about a potential attached debugger. Continuously, new methods are found to attack a specific technology. Thus the list of debugging attacks can never be complete. We shall have a look at some of those techniques in order to get an idea of how active anti-debugger techniques work.

Windows API functions and Flags: One of the simplest techniques is the use of Windows API functions like `IsDebuggerPresent()`, `NtQuerySystemInformation()` or `NtQueryObject()` (located in the system libraries kernel32.dll and ntdll.dll). The return values of those functions indicate a debugger in memory whereupon a well-fortified executable can react on the function’s outcome. Whereas the function `IsDebuggerPresent()` makes no secret of its purpose, other functions are less obvious. In the case of the latter two functions mentioned before they return a debug object which is created by Windows whenever a debugging session launches. Another important function worth mentioning here is `CheckRemoteDebuggerPresent()` which internally calls `NtQueryInformation-Process()`. Both functions can be used by a process as debugging defensive mechanism.

Flags as properties in the PEB[24] or in the the process’ heap are another source of information that can be used to determine if a debugger is present. If specific flags in the `NtGlobalFlag` field in the PEB are set to certain values, this may—yet not imperatively—indicate a debugger’s presence. The same applies to several heap flags that change their values dependent on the flags set in the `NtGlobalFlag` field.

[24]**P**rocess **E**nvironment **B**lock: A structure in memory which contains information about the associated process.

These techniques are described by Peter Ferrie in [16] and Mark Yason in [44]. Both also give more detailed information and examples on how these techniques are used.

Unhandled Exceptions: Another attack presented in [16] aims to trigger an exception by not setting up an appropriate handler. Whenever an exception occurs for which there was no according handler registered, the system finally catches it by calling the kernel32 function `UnhandledExceptionFilter()`. In turn this will result in a call to a fall-back handler that needs to be registered before by a call to `SetUnhandledExceptionFilter()`. However if a debugger is present, the exception will be passed to it and the fall-back handler will never get to be called. This way a program can infer the presence of a debugger if the fall-back handler is not called.

Timing Checks: When a process is being debugged, a reverser may halt the process and use the debugger to step slowly through the commands. A way to reveal this is to count the CPU cycles that are spent during execution. If the amount is far too high, then this is an indication that the process is being debugged. This technique is described in [44].

Debugger Windows/Debugger Process: By enumerating all process names, a defensible process may search for known names of debugger processes and react upon the detection of such. Also class names or title strings of windows are perfidious and can easily be detected by using a function like `FindWindowEx()`, [44].

Self-execution and Self-debugging: In order to fool the debugger and get away from its custody, a program may create two instances of itself as the debugger is only able to have control over the first process but not over the second one. The original process that is handled by the debugger creates a mutex and forks the second process. The latter realizes the presence of the mutex and therefore chooses a different control flow which will execute the actual program.

A related method is to let the second process attach to the first process as a debugger. As only one debugger can be attached to a certain process, ordinary debugging is not possible anymore. However, this attack can be defeated by using the kernel32 function `DebugActiveProcessStop()` to detach the debugging process or by setting the `DebugPort()` field in the EPROCESS data structure to zero. A description of these techniques can be found in [16].

Open process: An interesting technique that enables fortified programs to do more than just terminating themselves whenever a debugger is spotted is the following [16]: In order to obtain a handle on any process in a Windows system a certain process, mostly a debugger, must enable the *SeDebugPrivilege*. In doing this it also may gain full control of the system process CSRSS.exe. This privilege is even passed down to the debugger's child processes, thus its debuggees can exploit this situation by making CSRSS.exe perform some malicious operation. Concretely this operation may cause an infinite loop in a new thread of CSRSS.exe or the access to an illegal memory address which will both result in denial-of-service on system level. Conducting a privilege escalation attack by exploiting this vulnerability is only less reasonable as an already high-privileged administrator account is needed to get the SeDebugPrivilege.

Header entrypoint: A debugger can stop the execution of a debuggee at a certain instruction by setting breakpoints. When using software breakpoints the debugger replaces the instruction with an `INT 3` instruction. At execution it generates a software interrupt that tells the debugger to halt the program, restores the original instruction and lets the developer examine its state. Obviously, the section where the software break is situated and the `INT` instruction is written has to be writable which is normally the case for the `.text` section. Typically a default breakpoint is set at the first instruction of the program; if the entry point is located inside the PE header, however, a debugger might fail to set this first breakpoint there as the header's properties are undefined and therefore read-only. Sloppily implemented debuggers won't notice that setting the breakpoint failed and let the program run without control. Refer to [16] for more information.

TLS Callback: Presented in [16] another trick is to make use of TLS callback functions. A TLS (Thread Local Storage) of a certain thread contains its local private data that is not visible to other threads of the same process. A callback function can be defined to initialize this data. By default the debugger executes these functions before freezing the process at its main module's entry point. TLS callback functions offer a great opportunity to execute defensive code without the reverser to notice it. However, once this trick is revealed it can be easily countered by advising the debugger to break at the system entry point, identify the code in the TLS callback functions and set a breakpoint before them.

2.4.5 Anti-Dumping

At some point of time a packed executable has to be unpacked more or less into its original form to be executed. (This is not true for executables that were packed by an emulation obfuscator, cf. section 2.4.7 for more.) It is common practice to halt the program at its OEP and dump the program's image to disk which is called *manual unpacking.* Though in most cases some minor fixes need to be applied to make the executable work again, normally this is an effective method to undo a packer's work. Some counteractive tricks to prevent unpacking can be found in [16] which we shall have a look at in this section.

Nanomites

Nanomites are a clever and effective method to make it difficult to recover valid code, for dumping it onto disk. When furnishing an executable with nanomites conditional instructions are replaced by `INT 3` instructions as if breakpoints were set. The information needed to recover the original instruction is kept in the unpacking code. In order to let this program run the unpacker must also use self-debugging (cf. section 2.4.4), a method where two instances of the same packed executable run once as a debugger and a second time as a debuggee which does the actual job. Whenever a nanomite is hit, the debugging instance searches for information about this nanomite at this specific address in an address table. If a nanomite is registered at this address, the processor flags that need to be set for the right branch to be taken are read and compared with the currently set flags. If the pattern matches, a jump to the accordant branch needs to be taken. The address of the branch's first instruction is retrieved from a destination address table, a jump to this address is performed and the program goes on with execution. If the flags don't match, then the length of the conditional instruction that was replaced by the nanomite is found in the size table and execution goes on at the accordant offset.

Stolen bytes

The stolen bytes technique is essentially quite the same as outlining (cf. section 2.4.3) but chunks of code are moved to other places in dynamically allocated memory rather than being relocated in the program's image. Jump instructions lead to and from the replaced opcodes, whereas the addition of junk code at the addresses where once the stolen bytes were even increases the confusion factor.

Messing with the Import Table

A popular anti-dumping method is altering the import table once the IAT has been created. However, when a loader tries to resolve the dumped executable's imports, it fails to run it as the import table is broken or even non-existent. As a countermeasure there exist tools that help to restore a broken import table after dumping, with information from the IAT of the process which is still in memory. The IAT structure, however, may also be flawed, or non-standard respectively, as during unpacking the packer might have created a jump table in a dynamically allocated buffer that adds another level of indirection to the IAT.

Guard Pages

Instead of unpacking the whole program at once and then handing control over to the unwrapped executable, packers like Shrinker and Armadillo use on-demand decompression or decryption respectively. Unpacking becomes more difficult because the program is not restored in memory as a whole when it begins to execute. This techniques makes use of *guard pages* that trigger a STATUS_GUARD_PAGE_VIOLATION exception when a memory address within the guard page is accessed. Shrinker hooks the system's exception handler and catches any exception to the previously mentioned type, decompresses it and continues execution. Armadillo does the same but it uses self-debugging to again catch the exception. The 'good' thing from a reverser's view is that once a page is decompressed it stays in memory, so "by simply touching all of the pages in the image, Armadillo will decrypt them all, and then the process can be dumped entirely", as Peter Ferrie suggests in his paper [16].

Writing then Executing

Automatic unpackers try to guess the moment when the packer's stub is done with unpacking whenever a recently written page has executed. Anti-dumping code may thwart this algorithm by unpacking and executing dummy instructions that were interweaved with the actual code.

2.4.6 Anti-Emulation

A great application for virtualization is to create an insular environment where malware can be tested and watched safely without endangering the host system. If a system becomes irreversibly compromised or damaged, the system can be rolled back to a previous snapshot or entirely deleted. However, for an executable there are ways to detect if the environment it is running in is in fact native or if it is virtualized. Dependent on the outcome it may change its program flow. Some malware, for example, could refrain from doing any damage and therefore confuse the human observer who expects the specimen to evince malicious behavior.

In [14] Peter Ferrie proposes two different kinds of virtual machine emulators: *Hardware-bound* virtual machines directly let guest-code run on the real CPU whereas in a system of a *pure software* emulation type a complete CPU is emulated in software; it works "by performing equivalent operations in software for any given CPU instruction". In the case of the former, the CPU has to be shared between the guest and the host operating system which makes it necessary to use control mechanisms that support this configuration. The usage of these control mechanisms reflects in the environment which can be observed by an application that has knowledge about the typical patterns every implementation of an emulator leaves in the guest operating system. Based upon these observations the program may change its behavior. Usually software emulators aren't immune against detection either. As they are not perfect, slight deviations and incompleteness in the implementation of the emulated instruction set reveal that a guest operation system is actually emulated.

The ways an application can detect if its environment is currently emulated are highly dependent on the certain emulator's implementation that the application suspects it is virtualized by. To cover every detection technique would go beyond this paper's scope. The interested reader can refer to Peter Ferrie's work in [15], [14] and [16].

2.4.7 Emulation-based Obfuscation

The adoption of emulation as anti-reversing technique takes complexity as well as success within the meaning of anti-reversing to a whole new level. A weak point that all methods mentioned before have in common is the fact that sooner or later machine code of the original executable will appear in memory. Despite exhaustive security measures, it is only a matter of time and determination until conventional techniques of yesteryear can broken. So, either manually or automatically by an unpacker, the packed program can be dumped or at least analyzed. The effectiveness of typical unpacking approaches (discussed in section 2.5), however, becomes seriously restrained when *emulation obfuscators* come into play.

Overview of Emulation Obfuscators

Emulation obfuscators like Themida, Code Virtualizer[25] or VM Protect[26] allow developers to specify portions of a program's x86 code which will typically be one or more of the critical functions that need to be protected against reversing. These chunks of code are then translated into a custom machine language that is unequal to the original language that the program's native processor accepts. In order for the executable to remain fully able to run on its designated platform the packer furnishes the stub with an engine that takes care of the bytecode by interpreting it during runtime and executing the according x86 instructions on the native platform. The crucial point is that the original code is never restored again which makes it very hard to conduct both static and dynamic analysis. Obfuscation methods like those bring potency and resilience to a maximum level.

Usually the instruction set is different for every individual packed executable. For example the x86 equivalent for a certain byte `2Eh` may be `PUSH ESP` for one packed executable and `MOV EAX,EDI` in another instance. Of course, the stub's interpreter that comes with every packed executable is customized to fit the program's individual bytecode. Commonly the bytecode tends to have a reduced instruction set analog to RISC. So, when the CISC x86 instruction set is translated into a RISC-like virtual language it is broken down into more fine-grained instructions. So, obviously one x86 instruction will result in two, three or more bytecode instructions.

Confusion of ideas: Virtualization and Emulation

People have used several kinds of denotations when referring to the technique that we call 'emulation-based obfuscation' in this paper. Terms like 'virtualization', 'virtual machine' and 'emulation' are coined when this topic is given attention. Obviously it is not so clear how to use what term.

The naming of emulation obfuscators which implement emulation obfuscation (Code Virtualizer, VMProtect) or the terminology in certain papers or other documents may give the impression that some sort of virtualization technique is adopted. According to [31] a virtual machine is defined as "an efficient, isolated duplicate of the real machine". Goldberg states, it is a "system [...] which [...] is a hardware-software duplicate of a real existing machine, in which a non-trivial subset of the virtual machine's instructions execute directly on the host machine [...]", (qtd. in [22]). It is safe to say that the goal of virtualization lies in creating a replica of an existing machine that is comparably fast, executing opcode directly on the host's CPU, but is isolated in relation to other systems that reside on the same real machine as the duplicate.

[25] http://www.oreans.com/products.php
[26] http://www.vmprotect.ru/index.php?lang=en

According to Lichstein emulation is “a process whereby one computer is set up to permit the execution of programs written for another computer. This is done with [...] hardware features [...] and software” (qtd. in [22]). So, in both cases at least two computer systems exist within some hierarchy, whereas one computer executes machine code of the other. An emulator can be implemented on several levels; it may just emulate the operating system environment a program needs be executed (software emulator) or emulate an entire computer (PC emulator) that allows a whole operating system to be run. In this respect emulation and virtualization are similar.

The important difference is, however, that when it comes to emulation, the code is handled entirely by software and never executed directly by the CPU. Instead the emulator interprets the foreign opcode of the program and then translates it into equivalent native opcode. In opposition to that virtualized program code is passed down directly to the CPU. It is important to note that it has still no complete control over the entire CPU.

From this point of view it is intelligible as to why emulation obfuscators can’t be regarded as virtualizers, as it is in their very nature to interpret machine code that is diverse to that of the native machine. In fact there is always a stub, an emulator that is implemented in software and distributed with the protected executable which handles the obfuscated bytecode. This code is never and even cannot be interpreted by the real CPU. That is why it is feasible to define the technology described before as *emulation-based obfuscation.*

An example: Code Virtualizer

A reverser, known to the internet community merely as “scherzo”, has made the effort to delve into the internals of Code Virtualizer, a commercial protector of Oreans Technologies, and has published his findings in [34].

According to scherzo’s investigation Code Virtualizer follows a classic pattern that is also adressed by Sharif et. al., who have developed a generic unpacker for virtualization obfuscators in [37], cf. section 2.5.7. In this scheme the virtual opcode (often also referred to as bytecode) is processed by a main loop. It *fetches*, *decodes* and finally *dispatches* it to an appropriate handler which in turn finally *executes* native code and passes control back to the main loop which will begin a new cycle. Code Virtualizer consists of 150 handlers, all of which can uniquely be identified with an ID, and most of them execute the bytecode’s equivalent native opcode. Exceptions are handlers with the ID of `0x0E` and `0xA4` which are responsible for switching the context between the execution of native, regular code and the execution of bytecode. Managing the initialization and de-initialization of the emulation engine when needed is their job.

Increasing the complexity just by adding virtualization as another level of indirection is barely enough. To raise the bar Code Virtualizer creates bytecode that is different for every instance of a packed executable. This is achieved by introducing a random number that influences several components of the packed executable, namely the composition of the unpacking stub and the semantics of the bytecode. When a new executable is packed Code Virtualizer first writes the dispatcher and then all handlers in a randomized way dependent on the random number to the stub. To increase obfuscation each handler's code is stained with junk and then becomes interleaved as described in section 2.4.3. Furthermore a jump table is used by the dispatcher to direct the program flow to the right handler. To a certain degree this poses a good obfuscation rate in the packed program.

When Code Virtualizer is advised to virtualize a specific function's code this code is first disassembled, parsed and then translated into the protector's own intermediate byte format which is the same for every individual packed executable. This format is then the source for creating the unique virtual opcode. Essentially, amongst other factors, the virtual opcodes are used to calculate the right address in the jump table so that the accordant handler is executed. In addition to that, after a given handler is done with its job four instructions are processed that partly use random numbers to perform operations with values in certain registers. Together with the random number and the input bytecodes these instructions are used to calculate the address of the handler that will be called next. To make the confusion perfect, dependent on a random number, fake calls to handlers that globally do not change the state of the program are also interspersed.

Due to the random values used in this system, the bytecode issued by the obfuscator differs for every packed instance of even the same original executable and is only applicable for that very instance. Some bytecode interpreted by an unsuitable stub will cause the dispatcher to call the wrong handler via the jump table, make the program execute meaningless code and most probably crash it.

For more detailed information about the internal data structures used and a complete list of disassembled handlers refer to Scherzo's complete article with source code [34].

2.5 Comprehensive Counteractive Measures

While one side has developed efforts to enforce packed executables and make them robust against measures to break them, the other side developed ways to reverse the packing process and circumvent the packed security mechanisms. Obviously this is a rat race between those two parties; new anti-reversing techniques are being developed which finally will be discovered by malware analysts, who may then be able to bypass the obstacle with more or less effort and their newly gained knowledge. From the perspective of the latter, the ability to manually unpack[27] is not enough though.

As nowadays a large part of malware is packed, it must be possible to perform unpacking automatically. Implementations of this form, unsurprisingly called *unpackers*, make use of different kinds of algorithms and can be combined with antivirus scanners in order to detect malicious packed software.

It is reasonable to distinguish between *static* and *dynamic* detection methods. The former tries to gain usable information from the program's binary itself. For example traditional virus scanners compare malware signatures in their database with the ex-

[27] A great tutorial for manual unpacking and reversing explained from scratch can be found here: *Lenas Reversing for Newbies*, http://www.tuts4you.com/download.php?list.17.

ecutable's opcode and look for correlations. In contrast to this any dynamic analysis technique lets the program run and extracts information from the accordant process in memory. Usually the suspicious program is run in a sandbox that prevents it from doing any harm to the host system. If this is cleverly done the dynamic analyzer can extract much more useful information about the tested file than static detection methods are able to. However, one downside of dynamic analysis is the high complexity that makes successful development difficult. Because of the non-native environment in which the executable is run, dynamic analysis procedures take much longer than static analysis methods; scanning times up to 40 seconds are not uncommon. This poses another problem, as for many practical cases (anti-virus software that scans the system transparently in the background) this is not acceptable.

2.5.1 Signature-based Packer Detectors

Specialized detectors like PEiD[28] and virus scanners that don't support more sophisticated detecting methods are actually not unpackers. They are signature-based scanners for packed executables that statically determine if a given executable is packed and if yes, what packing algorithm was applied. As it is hereby not possible to determine the packed content, PEiD cannot give any information about whether the packed executable is harmless or harmful. Though detection rates are quite accurate regarding false positives, considering it is a signature-based system that relies on regularly updated signatures, PEiD is prone to false negatives, especially if an outdated version is used.[29] Although signature-based detection is fast, more advanced methods offer not only the detection of packed payload but also the feature of unpacking and subsequent malware detection—however, at the cost of a significantly increased scanning time.

2.5.2 Universal PE Unpacker Plug-in for IDA Pro

As a plug-in for IDA Pro[30] there is a simple unpacking tool available which tries to guess the OEP for a packed executable and halts the inspected process once the OEP is reached. The unpacker operates under the assumption that after the unpacking process is done, `GetProcAddress()` is always called to establish the import table and that afterwards the IP will finally be set to the OEP's address. The halting condition is met once the IP is set to an address within a certain range (which can also be altered manually by the user) in which typically the entry point is set for a PE file [11]. In this state it is assumed that

[28]PEiD: http://peid.has.it

[29]The terms *false positive* and *false negative* refer to the relation between the outcome of a test result and the actual condition, whereas the adjective "false" indicates the discrepancy between the two values. If the condition is given and the test result is negative, then is is a false negative as the test result is negative, but actually should have been positive if the outcome were correct. If the condition is not given and the test result is positive, then it is a false positive. In case of a signature-based scanner a false positive outcome denotes the detection of a malicious executable when in fact it is harmless and vice versa a false negative result is the detection failure of malfeasant code.

[30]IDA Pro: http://www.hex-rays.com/idapro/

the original program is unravelled in memory and can be dumped in order to restore the original executable.

While this unpacking technique works well for compressed executables, the heuristic explained before can easily be avoided by fortified instances as shown in [33] where the authors state that “several hundred malware instances sampled in the wild [...] evade the IDA Pro plugin’s heuristic”.

2.5.3 PolyUnpack and Renovo

A more sophisticated unpacker introduced in [33] is PolyUnpack. In this concept an executable’s code is statically analyzed and constantly compared to its executing corespondent process in memory. A potential packing mechanism is finally confirmed by PolyUnpack as soon as the IP points to code that was transformed by the preceding execution and therefore differs from the file’s code that was analyzed before. At this point it is assumed that the original code is about to be executed and the program is halted for dumping.

The authors could record a slightly better performance of detection rates in comparison to PEiD while PolyUnpack offers an additional unpacking feature. The big downside is, however, the extremely high processing time. Over 60% of specimens took less than five minutes, but 17 minutes was the average scanning time of all samples that were reported to carry packed code [33]. Others have confirmed its slow processing and criticized PolyUnpack’s inability to deal with multi-packed executables, i. e. executables whose code is protected by several layers of packing. Taking into account the evaluation in [33] and [19] PolyUnpack seems to have a satisfying detection rate but timeouts occurred when it tried to unpack executables packed with Themida.

Renovo [19] uses a similar approach to PolyUnpack but performs significantly better regarding detection speed. This is due to a different implementation technique: during static analysis PolyUnpack disassembles the inspected program’s code and stores each instruction for later comparison. When the program is executed Renovo disassembles every instruction the IP points to and checks if the instruction was recorded before. If this is not the case, an instruction that was not part of the analyzed program is about to be executed and the process is halted. Renovo doesn’t rely on disassembling but keeps track of memory addresses that have been written and subsequently marked as ‘tained’ by Renovo. If data at a tainted address is designated to be executed, then it is regarded as an unpacked instruction and the process is halted. The abdication of disassembling techniques is what makes Renovo eventually faster.

According to [19] it took 49.9 sec. on average to process each sample, while PolyUnpack was considerably slower with an average scanning time of 365.8 sec. As Renovo is capable of unpacking multiple-packed executables, it was able to unpack executables that PolyUnpack couldn’t.

2.5.4 OmniUnpack

OmniUnpack carries the approach of the latter two unpackers a bit further and implements an unpacking algorithm whose emphasis lies more on malware detection than on sole unpacking. As explained in [23] OmniUnpack keeps track of tainted memory pages that were written during runtime on the one hand and on the other hand of a subset of those dynamically written pages that contain instructions that were executed. If a dangerous system call follows an instruction that was written dynamically, then the program is halted and all tainted memory pages are scanned for malicious code. If malicious code is detected, then the process is terminated and its correspondent program on disk declared as packed malware. If not, then all marks of memory pages that were declared as *written* or *written and executed* are removed to prevent redundant malware scanning, the system call is allowed and the process resumes execution. If OmniUnpack encounters another suspicious system call, then the same procedure is performed again.

The authors “note that OmniUnpack does not need to observe all memory-page accesses. It is sufficient to observe the first memory access in an uninterrupted sequence of accesses of the same type. For example, only the first write to a page is useful, subsequent writes to the same page do not impact the result of the algorithm and can be ignored”.

Once again the newer system performs better than the ones developed before. OmniUnpack was evaluated together with PolyUnpack and outperformed its competitor regarding speed, effectively reducing “unpacking time from more then 100 seconds to 5 seconds”. Also while PolyUnpack failed to detect certain packed executables with a timeout of 300 sec., OmniUnpack was able to detect all of them within 10 sec.

2.5.5 Bintropy

Bintropy [21] pursues an interesting approach which tries to classify a given executable by statically measuring its degree of entropy; the calculated value may help to reveal if the checked executable is native or if it is packed. Entropy is the measure of uncertainty or disorder in a system or an array of bytes, thus entropy can reveal something about “the level of difficulty or the probability of independently predicting each number in the series”. If an executable gets compressed or especially if it becomes encrypted, then the amount of entropy of the file’s byte series rises, as it is has become appreciably more difficult to deduce a pattern within or any structure from these bytes. As described before, packers are also used to protect legitimate software against reversing, thus if an executable is classified as packed, this does not imperatively mean that it too is harmful and has to be contained. A classification system like Bintropy rather acts as a means for presorting potential malicious files by combining this system with a malware scanner. Presorting is important because unpackers tend to operate quite slowly and can increase the overall scanning time significantly. This is why it is desirable to use the unpacker for actually packed executables only.

The authors assembled a set of 100 executables from a default Windows XP setup and ran their tool on an unpacked, on a compressed and on an encrypted version of the file set. The result was a significant difference of the entropy value between native executables and the packed ones. However, the systems does not work precisely enough to clearly distinguish between compressed and encrypted executables. Also by deliberately inserting recurring byte series, an adversary can easily fool Bintropy and generate false positives that will be labeled as native executables which in fact they are not.

2.5.6 Perdisci, Lanzi and Lee's Classification System

The classification system proposed in [28] (subsequently referred to as 'PLL') has the same goal as Bintropy in separating packed from native executables but in contrast to Bintropy, which considers only entropy as measured feature, PLL extracts some more features before making a decision.

Like Bintropy PLL measures *entropy* in executable code sections, data sections and the PE header. As some packers also put executable code in the header, it is also reasonable to integrate the header's entropy into the outcome. The *amount of standard and unusual section names* is also taken into consideration as packers often add, replace or rename them to a packer specific name. For example, an UPX-packed executable typically contains two sections names `.UPX0` and `.UPX1`.

The *property-flags* of an executable's sections may also give clues as to the nature of the file: as the authors have observed, a packed file often doesn't contain any section that is marked as executable. Usually the `.text` section which normally contains the executable's code is tagged with an executable flag. Also, it is anomalous for a section to be marked as executable *and* writable, as normally executable code is never written during runtime. A deviation of the standard pattern may indicate a packed file.

Last but not least the *number of items in the IAT* is another feature to be investigated: While a typical executable of a middle-sized application has many dependencies on external libraries, which is reflected by the amount of IAT entries, a packed executable doesn't have as many. This is due to the fact that the unpacking stub is usually supposed to be slight and flexible. If it had to depend on external libraries, it would be very likely that the unpacking process would fail because of missing dependencies.

Compared to PEiD, which failed in detecting 30.8% of provided packed executables, PLL performed significantly better. PLL was tested with 6 different classifiers such as entropy threshold or naive bayes. Hereby it was possible to achieve a detection accuracy between 96.6% to 99.6%, dependent on the deployed classifier. Furthermore about 96% of those packed executables that were not detected by PEiD were found by PLL.

2.5.7 Defeating Emulation Obfuscators

As mentioned before in section 2.4.7 emulation obfuscators are a special case. The fact that an obfuscated executable's code is never unpacked in memory as a whole requires new methods for restoring the packed code. Unfortunately, research regarding automatic unpacking of these kind of programs has not yet progressed as far as one would wish for.

Still, there is a general approach for automatic unpacking. A prototype, Rotalumè, is presented in [37]: as pointed out by Rolles in [32] one crucial factor to successfully unpack a virtually obfuscated executable is to determine "the locations at which control flow enters the virtualization obfuscator". By doing this, it is possible to track the protected program's execution and restore native opcode that is equivalent to the original program. However, Rolles states that "[d]etecting entrypoints into the VM is a hard problem to solve statically [as] the control transfers into the VM look like any other control transfer". Rotalumè's technique consists of four consecutive steps each of which tries to extract specific information of one kind which is then used in the subsequent step. A key factor in trying to understand the emulator's layout is the assumption of the existence of "an iterative main loop that fetches bytecode based upon the current value of a virtual program counter, decodes opcodes from within the bytecode, and dispatches to bytecode handlers based upon opcodes" [37]. The ascertainment of this loop allows further crucial investigation to identify the data blocks within the executable that hold the bytecode and to collect information on how the bytecode is translated into opcodes and operands which enables one to draw conclusions about the bytecode's syntax and semantics. In order to do this, the specimen is executed in an emulated environment and studied in several steps to extract information.

The first step, called *Abstract Variable Binding*, is established to generate a map of areas in raw memory which are used as pointer variables. This is done by tracing the program flow and observing its read and write patterns. Forward binding identifies variables that are instated to read data from memory whereas the backward binding algorithm discovers variables that are utilized to write into memory. The relevance of this is to gather candidates for a Virtual Program Counter (VPC). This special variable acts as an instruction pointer for the virtualized program that always points to the next bytecode that will be *read* and finally translated into native opcode by the virtualization engine. In the case of a jump or a branch, the consecutive program-flow (which can be identified by the VPC being iteratively updated) is disrupted. So, in contrast to an 'ordinary' instruction which does not deploy a control transfer, the VPC's value is not derived by its preceding value but set, i. e. over*written*, by an independent value pointing to the address of the first virtual instruction of the branch that is taken. Both cases are covered by abstract variable forward and backward binding.

As a next step candidates for a VPC are determined meaning that the information gathered in step 1 is further analyzed at which clusters of memory addresses are created: n logged written or read memory addresses m_1, $m_2 \ldots m_n$ are embraced by a cluster C if they are pointed to by the same abstract variable p_m. If p_m is indeed the VPC then each memory address in C represents the location of some bytecode.

To determine the right cluster and its VPC, every cluster is investigated for a memory read pattern that suggest an execution of emulated code. This pattern constitutes the main loop described before that fetches bytecode following the VPC, translates it to opcode, dispatches it to an appropriate handler and updates the VPC. For each cluster it is assumed that memory-reads in it follow this very pattern thus several techniques are applied to verify this thesis: loop detection methods, cf. [1], applied to partial control-flow graphs (CFG) may reveal the main loop, while dynamic taint analysis acts as a tool to track the data flow through the assumed four phases—fetch, decode, dispatch and execute. Here on byte-level, data found in the memory ranges that is assumed to be bytecode, is tagged with an unique ID and furnished with a label that indicates what phase the byte is currently devoted to. For example a read instruction of a tainted byte *b* in the relevant memory region will cause the taint engine to tag byte *b* with a ‘fetch’-label and mark the accordant register that holds the pointer to this address as VPC. A subsequent detection of dispatch-like execution involving *b* replaces the current label with a ‘dispatch’-label and so on. The data collected up to this point is later on used to identify the syntax and semantics of the analyzed bytecode and to extract the control flow semantics of the program, used to create a CFG which in turn can ultimately be used for malware analysis.

The evaluation conducted on synthetic and real-life programs that were packed with typical virtualizers like Themida, Code Virtualizer and VMProtect showed that Rotalumè makes it possible to extract syntax and semantics from a virtualized executable. Still there are problems to overcome concerning Rotalumè’s more or less specialized approach. For example if the targeted virtualization engine does not work on the pattern ‘fetch, decode, dispatch, execute’ the techniques currently implemented will not be successful in any way or no better than partially effective.

3 Concept and Implementation of PPJ

In line of this thesis a novel polymorphic packer, called PPJ, was developed. Its purpose is to demonstrate the possible actions of a potential attacker in order to hide Java-based malware from state-of-the-art static virus scanners. While current Java programs or malware may be packed with packers such as ProGuard[1], according to our knowledge no Java-based packers exist which change the program flow and which are open-source and could be examined. PPJ aims to fill this gap and should be regarded as a means to explore what is possible and what A/V manufacturers should be prepared for in the future.

3.1 Overview

PPJ is a polymorphic packer/obfuscator for Java programs. It is implemented in Java. This packer is intended to take a Java program and transform it in such a way that the original functionality persists but the form, thus the program's source or byte code respectively, changes for every packed instance. Essentially PPJ implements a polymorphic[2] cryptor as described in section 2.2.1 and 2.2.2 in the Java programming language. The change of shape is achieved by taking certain parts of the program and transforming it into a form which is encrypted and later re-embedded into the original program code. The original code is removed and replaced by a stub which undertakes the task of transparently decrypting the relevant encrypted data with the embedded key, before invoking the original code. Because the encryption-key is generated randomly every time the packer is executed, the packed results will always differ from each other even though the same program was used as a source.

The packer was developed as a tool to research the reliability of current malware scanners that apply static signatures to detect malware and to explore the technical possibilities of Java in respect to obfuscation in this sense. Hence, the described obfuscation technique does not focus on impeding decompilers or human reverse engineers but it tries to restrain the effectiveness of static malware scanners which rely on signatures in order to detect Java-based malware. Of course, as a polymorphic packer, PPJ suffers from the same problems that those packers working on native opcode have. Once the packing algorithm is known an executable that was packed with a certain packer can be identified via its constant unpacking code and can be unpacked subsequently with a custom-tailored unpacker. This makes is possible to identify the unpacked payload as malware. However,

[1]**ProGuard:** http://proguard.sourceforge.net/

[2]In relation to an object-orientated language like Java the term 'polymorphism' can be confused with *subtype polymorphism*, which is, however, a completely different concept and has nothing to do with the packer's mode of operation.

up to now no packers of this kind for Java are known to the author, so it is interesting to test how malware scanners react to malware that was packed with a novel packer such as PPJ.

Besides thwarting static malware scanners, protecting intellectual property could also be seen as a potential application for the packer. Due to the nature of Java it is easily possible to decompile an executable that was already compiled to byte code, as the packer can be configured to encrypt valuable code which is not directly accessible after decompilation. Certainly with some effort the packing algorithm can be reverse engineered and the encrypted code can be retrieved as the key is included, but in combination with other packers that may focus on a different from of obfuscation a considerable amount of complexity and obfuscation can be achieved. The focus of this thesis however is the evasion of malware scanners, thus this kind of application is not further pursued here.

3.2 Related Work

Numerous packers already exist for Java. One of the best known is ProGuard, which is still under active development and published as open source under a GPL license. The website associated with ProGuard provides an elaborate list[3] of alternative packers to ProGuard. It differentiates between shrinkers, optimizers and obfuscators, where the different types may blur into each other. Shrinkers place their emphasis on removing unnecessary code, thus making the code smaller and faster to execute. As the name says, an optimizer's job is to optimize java byte code. Rather than removing dispensable code an optimizer analyzes the existing code and tries to re-implement it in a way that can be executed as quickly as possible by the Java VM. For instance classes that are not referenced by any subclasses may be declared as *final*, allowing the compiler to statically link this class. It does not need to provide infrastructure for dynamic binding at runtime which allows the code to execute faster.

Last but not least an obfuscator alters byte code in such a way that it becomes hard for either a decompiler and/or a human reverse engineer to retrieve and understand the original source code. In Java there are several ways this can be achieved. Most Java obfuscators are capable of replacing significant package, class, method or field names which are irrelevant to the VM in most cases. For a human however, descriptive names are a key to understanding what a part of a program actually does. Another popular approach is to insert `goto`-statements which, although not supported in the Java source code, are an important part of the byte code in order to model constructs like loops. Plain decompilers cannot handle randomly inserted `goto`-statements which do not constitute to any higher level construct. More techniques of the same kind are comparable to the ones described in 2.4.3. All of those methods have in common the fact that they *statically* alter the code, in contrast to PPJ where there is no dynamic generation of byte code.

Apart from ProGuard—which is capable of shrinking, optimizing and obfuscating—

[3]**ProGuard alternatives:** http://proguard.sourceforge.net/alternatives.html

there are obfuscators such as JBCO[4], JavaGuard[5] and Sandmark[6]; all three of which are also published as open source. The latter, developed at the University of Arizona under the supervision group of Christian Collberg and Gregg Townsend, is not just capable of obfuscating code but also incorporates algorithms for software watermarking and for code optimization. Software Watermarking is a process of embedding unique identifiers into the program code. Often this is done in order to be able to identify copies which were distributed illegally and to identify the original buyer. There is also commercial software such as DashO[7], Klassmaster[8] or SophiaCompress[9], which specializes in compressing Mobile Java.

3.3 Design

PPJ is a polymorphic packer in the sense of the definition in section 2.2.2. Parts of the packed program consist of normal program code, other parts consist of special packed code which is embedded as data within the program. This code is not immediately executable but after the program has started a certain part of the program code, the stub, is responsible for dynamically preparing the packed code during runtime in order to render it executable. All this happens in memory. After the embedded code has been prepared control is handed over to the unpacked code and it is executed.

A program that has been processed in such a way is able to hide the packed code to some degree from certain entities that have an interest in examining and understanding it. These entities may be a human or another program. As the packed code is treated as data the packer has the freedom to transform it in any reversible way. The data may have a different form in every example of a packed program while the set of instructions stays the same. By employing this technique the program is able to firstly hide code from immediate understanding and secondly evade simple static analysis. The downside of this approach is that the program always has to include the logic to decode the protected code. Thus understanding this part, which in contrast to the protected code is static for every instance, holds the key to also being able to understand the protected code. The trick is to make the unpacking code as complicated as possible to gain a big head start on reverse-engineers.

A popular way to employ this technique (though if implemented directly, an attacking reverse-engineer may quite easily see through it) is to use symmetric encryption. For every different key the cipher text, thus the protected code, has a different form which satisfies the necessary polymorphic property. Of course, the decryption code and the individual key have to be embedded into the packed program.

An important concept regarding polymorphic packers is that during runtime data gets transformed and is executed afterwards. Thus at some point a semantic change takes

[4]**JBCO:** http://www.sable.mcgill.ca/JBCO/
[5]**JavaGuard:** http://sourceforge.net/projects/javaguard/
[6]**Sandmark:** http://sandmark.cs.arizona.edu/
[7]**DashO:** http://www.preemptive.com/products/dasho/overview
[8]**Klassmaster:** http://www.zelix.com/klassmaster/index.html
[9]**SophiaCompress:** http://www.s-cradle.com/english/products/sophiacompress_java/index.html

place where code is interpreted by the CPU that was once treated as data. In fact the basic design of current computers (Von-Neumann architecture) does not differentiate between data and machine code in memory. The memory management of modern operating systems may segregate certain parts in memory from others in order to artificially implement this distinction for security reasons. However below the operating system's level, a CPU may treat any data in memory as opcode where the CPU's instruction pointer points to. Thus at machine code level—apart from the complexity and confusion one has to deal with while working on this layer—dynamic generation of code is easy. In essence, a `JMP` instruction at the right place at the beginning of the unpacked chunk suffices. As far as Java is concerned, dynamically executing code is not so easy to implement. In Java *class loaders* are used to load classes, thus executable code, into the VM during runtime.

PPJ operates at the source code level. That is, it accepts Java code as input and puts out obfuscated Java code. The user calling PPJ can specify which methods of the input shall be obfuscated. The selected function bodies are replaced by a stub which deobfuscates and then runs the unpacked code. As only the method's bodies are affected, the method's API stays untouched ensuring transparency according to the remaining code or code which is added in the future. Code which is not located in methods—code in constructors or in class initializers, as well as class fields—is not treated by the packer. In the following the concepts of PPJ are explained in more detail.

3.3.1 Dynamic code execution in Java

In Java the components which are responsible for loading classes are called *class loaders*. There are three built-in kinds of class loaders all of which adopt certain responsibilities, cf. [38] and [26].

"The *bootstrap* class loader built into the JVM [is] responsible for loading the basic Java class library classes. This particular class loader has some special features. For one thing, it only loads classes found on the boot class path. As these are trusted system classes, the bootstrap loader skips much of the validation that is done for normal (untrusted) classes." [38] The other two built-in class loaders are the *extension class loader* and the *system class loader*. The former loads classes from Java's extension APIs, the latter from the general class path which typically contains the application's class files.

Every application can define its own custom class loaders which may read and load class-files from other resources, for instance from a network. They all inherit from a class called 'Classloader' which the Java API provides and which allows a custom class loader to use its API. A custom class loader is encouraged to override `findClass()` in order to implement its own algorithm to obtain the bytes for a requested class. On the other hand 'Classloader' also provides `defineClass()` which actually loads a byte array into the VM creating a new class and returning a `class`-object. As a `final` method an inheriting custom class loader cannot override it. Furthermore it is deeply integrated into the specific VM's implementation and merely represents a stub for actual native code that is executed when `defineClass()` is called.

The union of all class loaders is strictly hierarchically organized: except for the boot class loader, every class loader holds a reference to its parent. When a given class loader is instructed to load a class that has not been loaded yet it first asks its parent to do this job, which happens recursively. If the class is not within the parent's scope then control is returned to the child which now tries to load it.

Additionally in Java the class loader is not only an entity dedicated to loading class files into the VM. It also influences a class' full name. In fact the namespace of a class is defined by its class-name, the surrounding package's name and also by the instance of the class loader that loaded the class. Assuming an application uses an external library which contains a class called `org.organization.core.CalcSerial` then it is likely to be loaded during startup. Program code which addresses a custom class loader to dynamically load another but different class with the same name into the VM is going to succeed, however instances of both classes are not compatible with each other while both having the same name (which can be revealed by calling `java.lang.Class.getName()`). This means that a class cast between both classes is going to fail because the original class was loaded by the system class loader and the fake one by the application's custom class loader, making the namespace unequal as the system class loader's instance and the custom class loader's instance are unequal. Note that two classes which originate from the same source (file, etc.) but are loaded by two different instances of the same custom class loader are also regarded as incompatible by the VM.

PPJ needs a functionality to dynamically load code during runtime. The system class loader, normally responsible for loading the application's classes cannot do this because it is not designed to load byte code from somewhere else but the stated general class path. The packer, however, requires program code which accepts byte code as a byte array (derived from the unpacked data) and loads it into the VM. Employing a custom class loader seemed to be the most appropriate solution to facilitate this (although during testing downsides revealed themselves for this approach, cf. section 3.5). As a consequence of this, every portion of code which is supposed to be packed is transferred into its *own* class; the smallest atomic entity which can be loaded by a class loader. For every packed method's body the code is moved into a single, so-called *outsourced* class with one static method where the code is inserted. When it is required, the whole class can be loaded during runtime and the single static method is invoked via reflection. In order to avoid name collisions the class' and the method's names are randomly generated.

3.3.2 Preparing code

In order to pack certain code the source code at first has to be identified and then modified. For PPJ a parser analyzes the class which contains the methods to be packed and translates it into an AST[10]. An instance of this AST consists of Java objects; the corresponding classes represent the entities of the Java syntax. For example there is a class for an `if`-statement, for a literal, for a field access expression or an `instanceof` key word. The objects are linked and depend on each other either by holding references to other objects or by using object-orientated concepts like inheritance or polymorphism.

The AST can be modified in order to change the corresponding source code that the

[10]**A**bstract **S**yntax **T**ree

AST stands for. This is necessary as the code is moved to the outsourced class in another namespace where the code isn't valid anymore due to invalid references: fields of the original class cannot be accessed anymore, the same applies to the other methods, nested classes or the object itself (`this`-pointer). In order to resolve this the outsourced class specifies a field which holds a reference to the original object. Via this reference it is possible again for the outsourced code to reach methods and fields of the original class. The reference is set during the unpacking process.

The second problem with outsourced code is that certain members of the original class cannot be accessed anymore due to insufficient access rights. The class' fields, methods and other members which are defined as `private`, `protected` or without access modifier cannot be accessed anymore from a foreign outsourced class in another namespace. Due to the restrictions described before in section 3.3.1 it is not possible to load the outsourced classes into the same package where the original class resides which would at least make it possible to access protected members or those with package visibility. Even worse, this also affects all other classes (neighbor classes) which are located in the same package as the original class. Their members may also be configured to being inaccessible by classes from other packages. Due to this the packer has to change the access modifiers of those members in the original class to `public`, that are accessed by the packed code. On the other hand for every class in the original class' package, the access modifiers of their members also need to be changed to `public` (with the exception of private members which could not be accessed before anyway).

When the above is done the outsourced class has to be brought into a form which can be modified in the sense of polymorphism and which can be re-embedded into the original class later on. As the custom class loader accepts a byte array to load classes into the VM it is reasonable to compile the outsourced class against the modified source code and use the out-coming raw bytes for re-embedding.

3.3.3 Making code polymorphic

Symmetric encryption is a solid technique for implementing the requirement of polymorphism. The output, thus the cipher text, of a good encryption algorithm features a very high amount of entropy. This means that encrypted code (which actually still contains the whole information of the plain text) is hardly or as-good-as undistinguishable from a random chain of bytes. As the form of the cipher text depends on the key a change of the key results in a different form of cipher text. For a good encryption algorithm even if the key only changed for one bit the cipher text is going to have a completely different form. This feature can be harnessed in order to change the form of every packed code while keeping the original code the same.

PPJ applies AES-256 to perform encryption on the byte array that the compiler yields for compiling the outsourced class. The key is appended to the encrypted byte array and then re-embedded into the original class.

3.3.4 Embedding the outsourced classes

The outsourced classes need to be unpacked and loaded into the Java VM at some point. The processed byte code needs to be stored at an appropriate place. Re-embedding it into the original class together with the unpacking code is a good solution. The stubs are responsible for unpacking and transparently loading the outsourced classes. They are also introduced into the original class.

For every packed method a new stub, based on a template, is adapted and introduced into the original class as a nested class. In order to avoid name conflicts the stub's name is randomly chosen. During the packing process the stub's only static field of type 'String' is set with the according Base64 encoded polymorphic byte code. Every stub consists of a static initialization method that is used to create a new singleton object of its own class and to provide it with a reference to the original class in order for the outsourced code to be able to call the methods and access the fields. Then there is a method that must be called to get the singleton object. When this method is called for the first time it performs the unpacking process of the corresponding outsourced class and loads the unpacked class into the VM before returning the stub's instance. For all following calls the code has already been unpacked and the stub's singleton is returned immediately. The third method invokes the outsourced class' only static method (which holds the unpacked code in its body) via reflection and passes possible arguments as array.

As for the packed methods in the original class, here the method-header is preserved in order to maintain the API however the body is replaced by a call to the appropriate stub. This code is generated and adjusted during packing depending on the method's arguments and on the return type. Firstly, an array of type `Object` is created and its slots are filled with the method's arguments. Then the outsourced code is called via the stub object. This call is wrapped in a `try`/`catch` block to handle possible exceptions. Finally, if the original method was not `void`, the return value is cast from `Object` to the appropriate type and is returned.

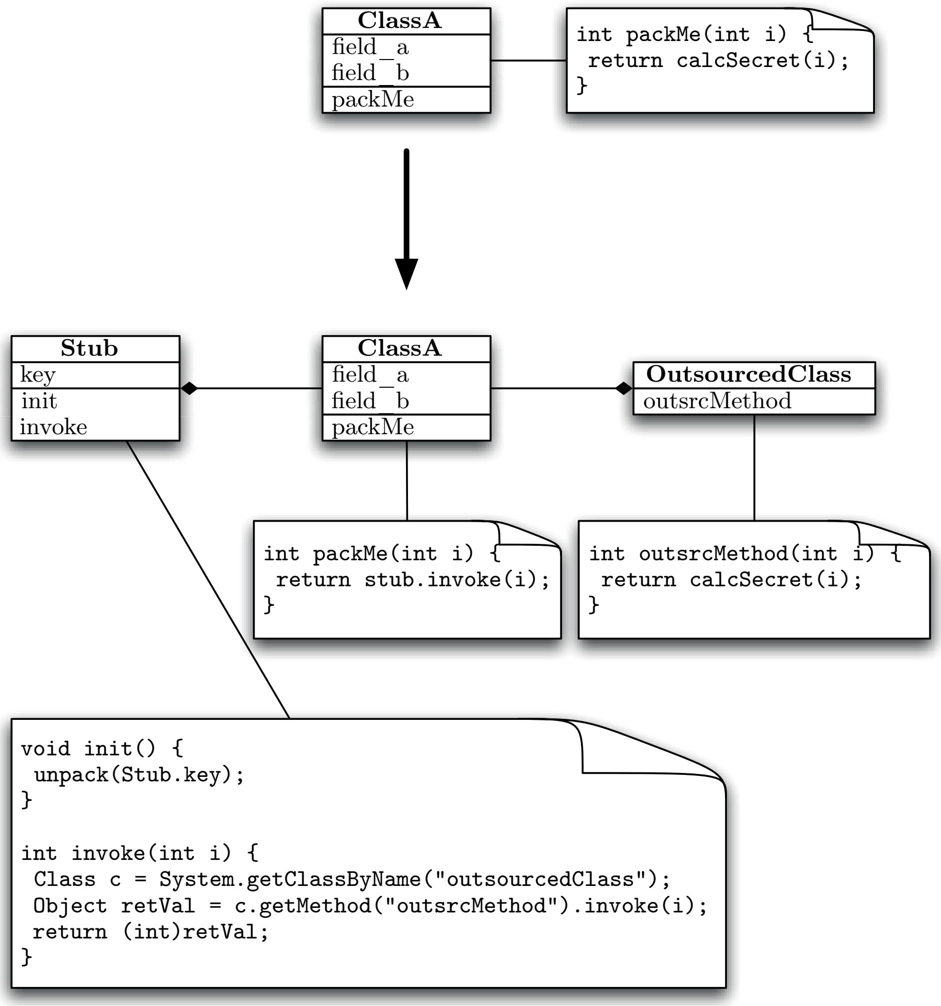
In addition to the stub some other helper classes are also introduced into the original class: among them a StubException class, a Base64Codec class (as the standard Java API does not implement any Base64 codec) and finally a custom class loader which allows the initialization and loading of a byte array constituting a Java class file.

3.3.5 Putting it all together – The workflow of PPJ

The packing process can be divided into several phases which will be explained in the following (q. v. fig. 3.1 and fig. 3.2).

Phase 0: Initialization

In an initial pre-phase the packer's configuration is analyzed. This means that the arguments' validity is verified, the existence and validity of referenced files is ensured and file system access checks for files and directories are performed. In the case that any of these requirements are not met, the packer exits with an error message on stderr.



```
int packMe(int i) {
 return calcSecret(i);
}
```

```
int packMe(int i) {
 return stub.invoke(i);
}
```

```
int outsrcMethod(int i) {
 return calcSecret(i);
}
```

```
void init() {
 unpack(Stub.key);
}

int invoke(int i) {
 Class c = System.getClassByName("outsourcedClass");
 Object retVal = c.getMethod("outsrcMethod").invoke(i);
 return (int)retVal;
}
```

Figure 3.1: The diagram shows the transformation of a class into a packed class in a simplified way. The original class' single method, 'packMe' should be packed. The result is a class with the same API but with an altered method body and two additional references; one to the stub and one to the class which contains the packed code in plain text. While the original body contained some secret calculations of a number, now the body of the packed method contains a call to the stub's method 'invoke'. In turn this method gets a handle for the outsourced method from the system and invokes it via reflection.

Phase 1: Preparation of packed methods

In the first phase the relevant Java source code file is read from disk, is parsed and is then used to compile an AST. The methods which were declared in phase 0 are determined and their code is being isolated. This code is modified so that it is going to be valid when it becomes inserted into a new class. Mainly references are added or bent to point to a

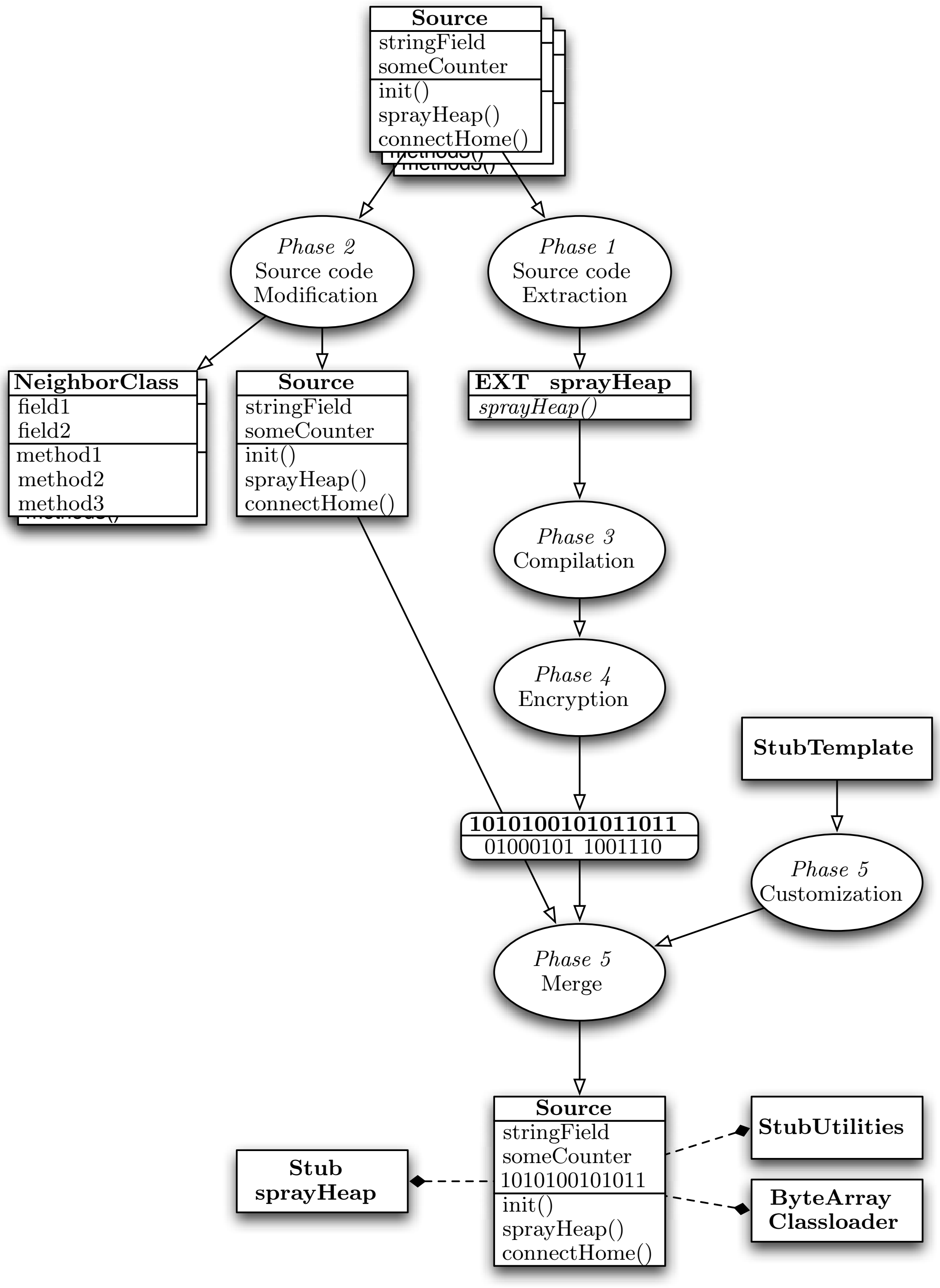


Figure 3.2: The packing-workflow: The code of the selected class' method (`sprayHeap()`) is extracted and put into an outsourced class with one static method that corresponds to the method in the original class. Then the outsourced class is compiled, encrypted and encoded into Base64. In the next step the original class is merged with a customized stub for every packed method and the encrypted outsourced class(es), which constitutes the packed method's code. The final packed class provides the same API, however the code of every packed method is now replaced with the stub which transparently unpacks the actual code (which now resides as a new string field in the packed class) when required.

local static field which is going to hold a reference to the original class.

Phase 2: Changing access modifiers

As the packed code is going to be moved to another class in another package, private and protected members of the original class or those with package visibility cannot be called anymore. Those members of the original class which have a different access modifier than public ones and are accessed by the packed code need to be changed to public. This is done in phase 2 by modifying the AST which has already been built in phase 1.

The same applies to 'neighbor' classes in the same package. Their members could be declared with package visibility or protected and thus cannot be accessed anymore by the outsourced code either. Also for those members the access modifiers need to be switched to public.

Phase 3: Compilation of the outsourced classes

In the next step the outsourced Java code needs to be compiled to byte code which can be loaded into the VM. However, the compilation is going to fail if the code is compiled against the original source due to the invalid access modifiers as described in the previous paragraph. Before the compilation can be performed the modified source class and all other modified classes in the same package also need to be compiled. After this has been done the resulting byte code is passed to a custom class loader, called *InterceptionClassloader*. This class loader breaks with the stipulated calling sequence as described in section 3.3.1 where a child class loader always asks its parent to load the requested classes. As the name says the InterceptionClassloader first takes a look if it holds the byte code to a requested class and loads it into the VM if this is so. This way the compiler does not take into account the old class files from disk (with the invalid access modifiers) but the freshly compiled classes when the outsourced classes are compiled. Thus the compilation of the outsourced classes succeeds.

Phase 4: Encrypting and encoding the outsourced byte code

After the outsourced classes have been compiled every resulting byte array needs to become encrypted and brought into a form which can be stored together with the according stub. For this the packer randomly generates an AES-256 key and encrypts the byte code with it.

The encrypted data cannot be embedded into the stub alone; more data must be added so that the packed code can be unpacked and executed. The following information must be combined:

- The encrypted byte code
- The class name(s) of the outsourced class(es)
- The key for the encrypted data

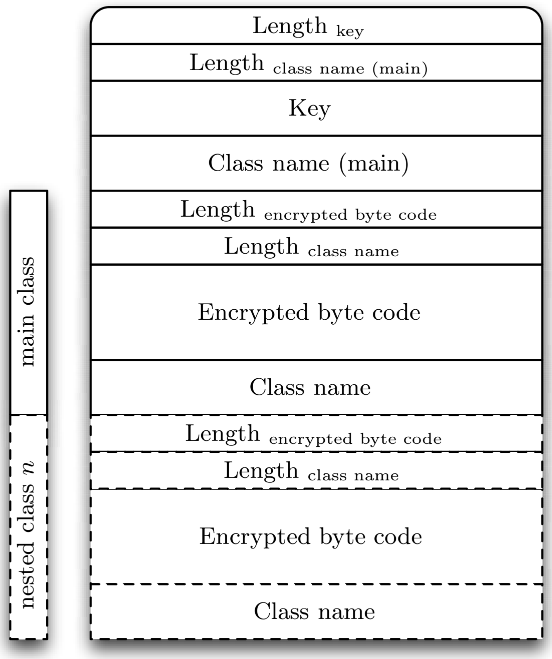

Length $_{\text{key}}$
Length $_{\text{class name (main)}}$
Key
Class name (main)
Length $_{\text{encrypted byte code}}$
Length $_{\text{class name}}$
Encrypted byte code
Class name
Length $_{\text{encrypted byte code}}$
Length $_{\text{class name}}$
Encrypted byte code
Class name
main class
nested class n

Figure 3.3: The diagram shows the layout for the byte array which is embedded into each stub. It always starts with the length of the key and the length of the outsourced class' name; both values have a fixed size of 4 bytes. They are followed by the key itself and the class name. 4 bytes are stored twice, which contain the length of the encrypted byte code and (again) the class name's length. Again the actual data is followed. Although it is not currently implemented an outsourced class could have some nested classes which would result in separate files when they are compiled. The scheme is prepared to embed them after the main class.

All of this is moved into a new byte array which is structured in a certain way, shown in figure 3.3. In order for the parser to know how many bytes to read, the actual data is preceded by a fixed size of bytes which contain the length of the data. In this case the length of the key and the length of the class name is stored in two groups of 4 bytes each, followed by the key itself and the class name string. Although it is not currently implemented, the outsourced class could have x nested classes which would result in $x+1$ separate byte code entities when compiled. In order to provide for this the structure allows the storage of an arbitrary number of encrypted byte arrays of byte code. After the class name's string the length of the following encrypted byte code and the length of the class name is stored; again in two groups of bytes by 4. After this the encrypted byte code and the class name is stored. This is repeated for every nested class.

Once the byte array, which accumulates all data, has been created it is converted into Base64 which allows the storage as a string. Later this string is stored as a constant, as part of the stub.

Phase 5: Embedding the stub with the packed code

Finally, the packed code has to be embedded into the original class together with the stub. A stub template file is used to create a skeleton AST for the stub which is completed in order to fit the specific packed method. On the one hand the Base64 encoded byte array which was created in phase 4 is added as a constant in the stub. Also, the number of passed arguments and the return type of the packed method are customized, as well as the exception which can be thrown by the outsourced method and which needs to be caught.

For every packed method one stub class is added to the original class (as nested class), as well as other classes. Among them the byte array class loader which loads the unpacked outsourced classes into the VM. The method bodies are then replaced by an appropriate call to the respective stub.

Phase 6: Writing the class files

In the final phase the ASTs for all classes (the modified original class and all neighbor classes) are translated back into actual Java code and written to files. For this purpose a new directory which corresponds to the package that contains the original class is created. To avoid name conflicts an underscore as prefix and a running integer as postfix is added to the directory's name.

3.3.6 The workflow of a packed file

PPJ produces Java source code which can be compiled by a Java compiler or modified in another way before compilation. A state diagram of the workflow is shown in figure 3.4. An application with one or more packed methods may be launched like the original application, as the API was not changed by the packer. When a packed class is loaded into the VM all the stubs in this class are initialized. This means that a singleton is created and a reference to the original class' (`this`-pointer) is stored in the stub. This reference is passed on to the outsourced class instance when it is unpacked.

When a packed method is called, the accordant stub is instructed to invoke the only method of the outsourced class which contains the actual code. If the method has not been called before the outsourced class is still unpacked and not yet loaded into the VM. The stub notices this and invokes the code which performs the unwrapping. At first the stub's string constant, which contains the packed code in Base64 format, is read and converted back into a byte array. The byte array is parsed and all data (the key, the class names, the encrypted bytes) are stored in a data object which is referenced by the stub from now. The encrypted classes are decrypted with the key and are then passed on to the byte array class loader (which has been embedded along with the stubs) together with the classes' names. After this the class loader loads the outsourced classes into the VM.

Once this is done, the stub can proceed with the normal program flow which is always followed for all future calls of this method. In order to invoke the code the class object of the outsourced code has to be retrieved by passing its qualified name to the system.

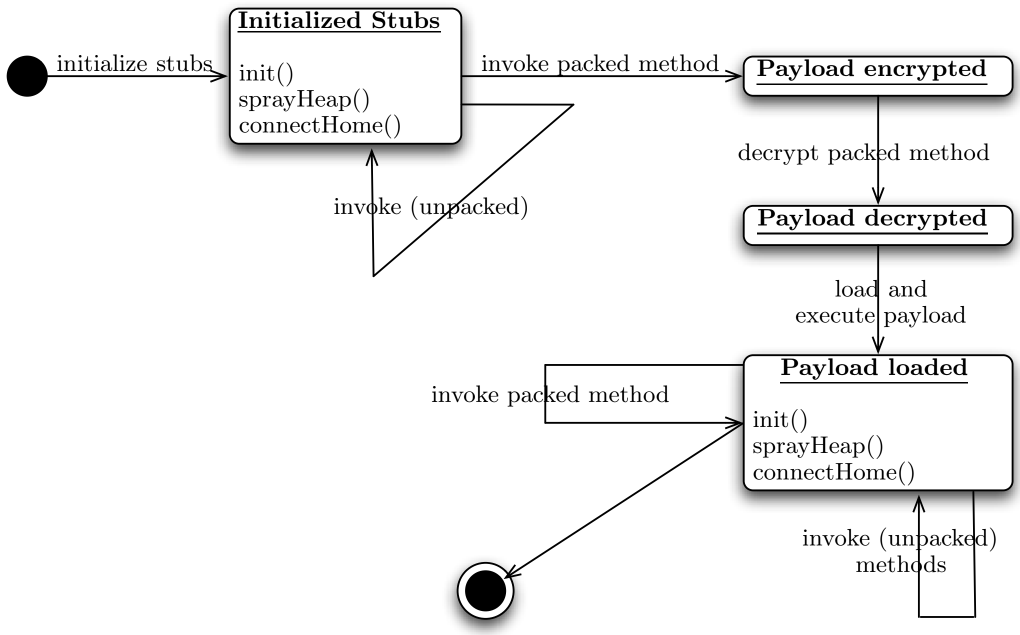


Figure 3.4: The state diagram for invoking a packed method: After the class (and all stubs) have been initialized a packed method remains unpacked as long as other unpacked or normal methods are called. When a method is invoked which has been packed but has not been unpacked yet then the stub will primarily decrypt and load the packed method's code and finally invoke the unpacked code via reflection. For each additional invocation the code will be called directly as it stays in memory.

This name can be fetched from the data object which was created during the unpacking process. An array of all methods in this class can be retrieved from the VM at which the length of the array is always going to be 1 as every outsourced class has been created with one only static method which contains the packed code. In the following the method is invoked with an array of type `Object` which represents the passed arguments. The method returns an object of type `Object` which can be cast to the original return type unless the method was not `void`. This cast object is then transparently returned by the stub to the caller of the packed method.

3.4 Implementation details and difficulties

In this section important implementation details are specified. PPJ strongly depends on Janino[11], which is an embedded compiler. "Janino is a compiler that reads a Java expression, block, class body, source file or a set of source files, and generates Java bytecode that is loaded and executed directly. Janino is not intended to be a development tool but

[11]**Janino:** http://janino.net/

an embedded compiler for run-time compilation purposes, e.g. expression evaluators or 'server pages' engines like JSP." [41]. Janino offers many features which have been useful for the creation of PPJ. In addition to the compiler there is a Java parser and a class infrastructure which allows the building of an AST and enable it to programmatically modify the AST. A traverser which can walk through the entire tree is implemented as *Visitor Pattern*. Thus, developers can create their own classes which inherit from the root traverser and change the properties of a node or rebuild parts of the AST in the way that the developer intends.

Some difficulties in Janino's design make its usage complicated. On the one hand it is not possible to exchange a node in the AST with another node and re-use the child nodes of the original one. When a newly created node is inserted into the AST it is initialized with the scope that the node ranges over. Once this scope has been set it cannot be reset anymore; the design of Janino prevents this programmatically. Therefore, if a node is replaced, all other child nodes beneath this very node also need to be recreated as the old nodes cannot be reinserted into the AST anymore. This slows down the execution speed, especially if many nodes need to be replaced and if these nodes reside in high positions of the AST.

Janino allows the execution of Java code fragments, like such as `(56 + 2) * 5.3`' or '`System. out.println("Hello World!");`. Internally it puts the code into a new class, compiles it and finally executes it. PPJ is based on this scheme, so it would make sense to reuse Janino's code. The nuisance is that, although Janino is designed in a very modular way and would perfectly support code reuse, all relevant classes are declared as `final`. In Java if a class is declared as final it is not possible to inherit from it anymore. As the compiler doesn't furnish such classes with code for virtual methods which can be overridden in a sub-class, the methods can be linked statically in favor of execution speed. The downside is that these classes lose the feature of an object-oriented language which allows the extension of code (which is actually desired sometimes to prevent code from being altered). However, this is exactly what was needed in this case. The only way to use this code was to create separate classes with altered copies of the original code. However, from the viewpoint of a proper software design, in no way is this a good solution to extend code.

As for the cryptographic functions which are used in PPJ, it accesses cryptographically strong algorithms of the standard Java extension API. In fact this is not urgently necessary for the requirements of PPJ, however it follows best practice. The random names are retrieved via `UUID.randomUUID()`, the keys are generated using a `SecureRandom` object.

Originally PPJ was intended to implement a packing technique similar to emulation-based obfuscation as described in section 2.4.7. However, after the development started it soon turned out that this project was going to be too complex to implement as part of this thesis. Thus, the project goal was changed to implement a packer with polymorphic properties. The building of an efficient java- and emulation-based packer depends on future efforts.

3.5 Limitations

Due to several reasons PPJ suffers from some limitations which are described in this section.

As PPJ is based on Janino it inherits all the limitations that Janino has. This concerns certain Java constructs which the parser is not able to process. They are specified on the homepage of the project [41].

- `assert`: a language feature which allows the developer to check if a certain assumption about the program state is met
- `enum`: Java built-in typesafe enumerations
- Parametrized types, generics
- Enhanced for loop; e. g. `for (element : elemArray) {}`
- Variable arguments; e. g. `static int sum (int ... numbers) {}`
- Annotations; e. g. `@Override public String toString() {}`

Whenever the parser encounters one of these constructs it is going to fail in constructing the AST. Besides, Janino's parser is unable to resolve some identifiers. For example, if it encounters a statement like `andy.read(that, book);`, where 'andy' is a field of the local class, it is unable to resolve the identifier `andy`, nor can it resolve the types for `that` and `book`. The latter would have been necessary to unambiguously identify a called method for changing its access modifier, the former to resolve an accessed field. It is up to the developer to change the statement to `this.andy.read(that, book);` in order to specify that a local field is meant.

Furthermore PPJ is not able to pack code which references members of a super class, hence code that contains the key word `super`. As the packed code gets outsourced into a completely different class, referencing the super class is futile in this context. Although it is possible to assign a reference of the current object to a reference variable it is not possible to do the same for an object of the super class. `Object obj = super;` is not going to compile while `Object obj = this;` works perfectly. Thus it is not possible to circumvent this problem as if it was solved with `this` by passing a reference to the original object to the packed code. A workaround is to create an own method which is not packed and which is called by the packed code and merely performs the call to the super class' object. This workaround has to be written manually as the packer itself is not capable doing this.

As described in 3.3.2 the access modifiers of some members of all classes in the package where the original class resides need to be changed to public. As these key words don't just enforce *encapsulation* as an aid for developers to produce tidy, maintainable code but are also used to implement security concepts, the change of access modifiers may also weaken the security concept of an application. For a final application this will not pose a threat as the packing is going to be done as one of the last steps in a development cycle, after the application has already been completed. This is different for libraries of applications which provide support for plug-ins. Dubious add-ons could subvert the integrity by setting internal data of a process to an inconsistent or invalid state. The same applies to libraries which are utilized for third party applications and could be misused by accessing members which were not intended to be used by external code.

It is well known that most packers which follow a polymorphic design can easily be automatedly unpacked once the algorithm is known, as it is generally based on security through obscurity. The same applies to PPJ. However, it should be kept in mind that PPJ is a proof-of-concept which is intended to explore packing techniques in Java and does not claim to implement the most sophisticated packing algorithms that are technically possible.

4 Evaluation

In this section we evaluate our prototype and demonstrate its usefulness in real-world experiments. Specifically we show that our prototype meets the following three properties:

- **Functionality:** The packer should not alter the functionality of the original program.
- **Efficiency:** The overhead that the packer imposes on the original program should not slow it down noticeably.
- **Obfuscation:** The packer should provide enough obfuscation so that preferably no static malware scanner is able to detect any malware which was packed with PPJ.

The main focus of attention is on obfuscation, as the approach was to analyze how static malware scanners can be circumvented. In the following section this will be discussed in more detail.

Functionality was tested during development of PPJ along the way with the evaluation of obfuscation and efficiency. Unless a formal verification of a program is made a developer cannot guarantee that the program does not contain any more bugs. A formal verification is a very complex task and would have gone beyond the scope of this paper, however, considering PPJ is a prototype and informal testing is a common reasonable measure, the tests conducted on the packer should suffice. Besides minor imperfections and shortcomings, some of which are described in 3.5, the packer produces valid code that corresponds to the original functionality. This was also assured during the efficiency tests, where no program behaved differently after having been packed.

4.1 Evaluation of Obfuscation

According to Barak [2] obfuscation is one of the conditions that a packer has to fulfill. It refers to the effort which is necessary for a human to understand the code and to recover the original one. In the case of PPJ it is more interesting to know the degree to which the packer is able to thwart static malware scanners. As our packer adds some not insignificant amount of code when obfuscating method-bodies, a static malware scanner should not be able to detect packed malware anymore.

4.1.1 Test Preparation and Setting

In order to test the performance of virus scanners on packed malware it was necessary to obtain Java malware which would be identified as such by most scanners, which could then be re-packed with PPJ to compare the scanner's performance with the packed and unpacked version. Rapid 7's[1] tool 'Metasploit Framework', an open penetration testing tool which allows security professionals to test certain environments for their security, served as one source for these programs. Metasploit Framework contains a set of exploits which may be used to attack vulnerable applications, services, etc. Among them are exploits for older versions of Sun's Java implementation, which allow breaking out of the Java sandbox of a Java-Applet. However, Metasploit Framework contains only a limited number of suitable exploits; a local producer of anti virus software kindly provided us with more malware samples, some of which we were able to employ for the evaluation.

A difficulty for this evaluation was the fact that PPJ operates on source code, which is utmostly difficult to find for actual malware. Fortunately, the nature of Java allows one to decompile byte code relatively easily and enables us to reconstitute source code that is almost equivalent to the original one. This approach turned out to be very effective for the samples from Metasploit. All of them could be successfully decompiled into valid Java code that also executed flawlessly after re-compilation, when we verified the exploit's functionality. The second set, which we obtained from the A/V company consisted of samples from 'the wild', which led to some complications. Obviously, all of them were packed so a successful decompilation was not possible for some of them in order to prepare them for packing. Additionally, some samples constituted only fragments of a complete program, so we managed to prepare only one for execution with its original functionality. We used two different decompilers, JAD[2] and JD[3]. A combined approach for decompiling proved as effective, as there were many cases where one decompiler was not able to do a successful decompilation, but the other one produced valid Java code.

As a reference for malware detection we used the service at VirusTotal[4]. It allows the user to submit files which in turn are checked against 43 static malware scanners which run on the servers at VirusTotal. After all checks have finished the website provides the user with statistics about which scanner has found the file to be malicious, and what kind of malware has been found. It is up to the user to interpret the results.

For every malware sample we conducted the following steps:

1. Decompiling all classes that constitute the malware back into valid Java source code. We incorporated only those samples for which this step was possible.

2. Re-compiling all classes and integrating the compiled files into a jar-file.

3. Scanning the re-compiled classes with VirusTotal by uploading the jar-file.

4. Packing all methods of all decompiled classes.

5. Re-compiling all packed classes and integrating the compiled files into a jar-file.

[1]**Rapid 7/Metasploit:** http://www.metasploit.com/
[2]**JAD - the fast JAva Decompiler:** http://www.varaneckas.com/jad; the original site, http://www.kpdus.com/jad.html, is not available anymore
[3]**JD - Yet another fast Java decompiler:** http://java.decompiler.free.fr/?q=jdgui
[4]**VirusTotal:** http://www.virustotal.com/

Name	**detection rates [re-compiled]**	**detection rates [packed]**
Java.Backdoor.ReverseBackdoor.A	9%	0%
Java/Agent.HC	2%	0%
Trojan-Downloader.Java.Tinconc	5%	0%
Trojan-Downloader.Java.Rebhip	2%	0%
Virus.Java.Djewers	12%	0%
Trojan-Downloader.Java.Agent	21%	2%
Trojan-Downloader.Java.OpenStream	20%	0%
Java.Trojan.Exploit.Bytverify.J	2%	5%
Trojan.Java.Downloader.P	10%	7%
Exploit.OSX.Smid	37%	0%
Exploit:Java/CVE-2009-3869.N	7%	0%

Table 4.1: Here the divergence of detection rates on VirusTotal between re-compiled and packed malware is shown. In almost all cases the detection rate was lower for packed samples and in most cases the packed binary could not be detected anymore by any A/V scanner featured at VirusTotal.

6. Scanning the packed classes with VirusTotal by uploading the jar-file.
7. Comparing the result from the first VT-scan with the result from the second VT-scan.

4.1.2 Test Results

We managed to perform the process described in the previous section for 11 malware samples. In order to identify all malware each sample was given a unique name. The name in our evaluation is determined by the majority of A/V manufacturers which specify the same name for one given malware sample on VirusTotal. The statistics in table 4.1 show that the initial detection rate for unpacked[5] re-compiled samples remains on a very low level; 37% being the highest and 2% being the lowest value. This already indicates a fairly poor performance of virus scanners in general for unpacked known samples, which were decompiled and re-compiled. For almost every case the detection rate dropped to 0% when they were packed with PPJ.

[5]In this case 'unpacked' refers to a program that wasn't packed with PPJ; it does not indicate whether a malware sample has already been packed with another packer before.

Table 4.2 gives a more detailed view of our evaluation result. In some cases one or more samples could be detected even after the malware was packed. The cause for this could be tracked down to a static field called *serialVersionUID*. This is an optional special field used by Java when object serialization is assigned. With object serialization an instantiated class can be flattened to byte code for storage on media or transmission over a network. At a later moment the serialized object can be deflated on another Java-compatible system and may be incorporated again in a running process. For this the class-type of the serialized object must be known. However, it may happen that on the system where the object becomes de-serialized a different version of the corresponding class is in used than on the system where the object was created and flattened. In order to handle this case properly every class that is marked as serializable should be supplied with a private static numeric constant called serialVersionUID which uniquely designates a classes' version so that the Java subsystem is able to identify incompatibility issues with serialized objects on different systems with different runtimes/libraries. Some malware samples contained such a field which is used by some A/V vendors—especially Sophos—to identify certain malware. However, considering the field's purpose a (malware) developer may as well change or even omit the serialVersionUID. In cases where malware scanners could identify packed malware by this field, a packed version of the concerned malware with the field commented out could not be identified anymore.

Some samples could not be decompiled properly into valid Java code. In three cases the syntax errors were minor and could be corrected manually. They are marked in a separate column in the table.

The detection performance of individual search engines varied. While Sophos detected nearly every malware sample before and some even after they were packed, other engines were rather disappointing; among them Kaspersky, Microsoft and Symantec. In the middle were Bitdefender, NOD32 or F-Secure and G-Data.

Overall, one can attest to PPJ a good performance, in respect of obfuscation. In 8 out of 11 cases it was possible to pack the malware successfully, meaning that 0 of 43 virus engines could not identify it as malware anymore. In two other cases a manual removal of the field serialVersionUID rendered the packed malware undetectable and in one case the packed malware without serialVersionUID was still identified by 2 of 43 virus scanners.

4.2 Evaluation of Performance

Another interesting property of PPJ is performance. We aim for an unpacking process that does not have a significant impact on the execution time of the packed program. Looking at the implementation details of PPJ we can point out two different spots in the transformed program where our packing mechanism leads to performance overheads:

- Once, the first time, a packed method is called
    1. Decoding the Base64-coded string.
    2. Decrypting the externalized class.
    3. Loading the externalized class into the VM.

Name	Status	AntiVir	Avast	Bitdefender	F-Secure	G-Data	Ikarus	Kaspersky	McAfee	Microsoft	NOD32	Symantec	Sophos	detection rates	manual src.-correction
Java.Backdoor.ReverseBackdoor.A	Re-compiled			●	●	●								9%	●
	Packed													0%	
Java/Agent.HC	Re-compiled								●					2%	
	Packed													0%	
Trojan-Downloader.Java.Tinconc	Re-compiled										●		●	5%	
	Packed													0%	
Trojan-Downloader.Java.Rebhip	Re-compiled												●	2%	
	Packed													0%	
Virus.Java.Djewers	Re-compiled			●	●						●		●	12%	
	Packed													0%	
Trojan-Downloader.Java.Agent	Re-compiled		●	●		●			●				●	21%	
	Packed												●	2%	
Trojan-Downloader.Java.OpenStream	Re-compiled		●	●	●	●					●		●	20%	
	Packed													0%	
Java.Trojan.Exploit.Bytverify.J	Re-compiled							●					●	5%	●
	Packed							●					●	5%	
Trojan.Java.Downloader.P	Re-compiled	●					●						●	10%	●
	Packed	●					●						●	7%	
Exploit.OSX.Smid	Re-compiled		●	●	●	●	●			●	●	●	●	37%	
	Packed													0%	
Exploit:Java/CVE-2009-3869.N	Re-compiled						●						●	7%	
	Packed													0%	

Table 4.2: This table shows the relationship of detection rates on VirusTotal between re-compiled and packed malware and the performance of individual anti-virus detection engines. Note that only a subset of all A/V manufacterers that are incorporated on VirusTotal are shown in this table; the detection rates refer to the entire set of A/V manufacterers on VirusTotal. The last column marks malware samples which couldn't be decompiled properly but could be manually corrected.

- Every time a packed method is called
  1. Executing the method's stub.
  2. Invoking the unpacked method via reflection.

At a first glance decrypting the externalized class seems to be the most performance-consuming operation that was added during the packing process. While this is probably true, one has to consider that the decrypting only happens once, namely the first time a method is called after the process is started and is of only minor consequence in respect of overall execution time. In fact, the invocation of a packed method via reflection is the real bottleneck if a method gets called a lot of times during a process' execution. This is shown later on with certain experiments with packed Java programs.

4.2.1 Test Preparation and Setting

The performance tests were conducted on following system:

Apple MacBook
Intel Core 2 Duo, 2.2 GHz
3GB RAM
Mac OS X 10.6.7
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334-10M3326)

In order to show the performance losses when a program gets packed we examined three Java programs. One program, *PiCalc*, calculates π to a configurable accuracy, another, *PrimeFactors*, performs prime factorization and a third one, *EncDec*, encrypts and decrypts and arbitrary strings with a configurable amount of iterations. We describe each of the three programs in the following paragraphs. The source code for PiCalc was taken from the website *</dream·in·code>*[6] and more code was taken from a blog called *Solvium*[7] for PrimeFactors. Both programs have been modified to test aspects of performance.

PiCalc

PiCalc implements *Leibnitz' formula for pi* [5], which can be displayed in various notations:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4} \tag{4.1}$$

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}. \tag{4.2}$$

The program's core consists of a loop whereas the amount of its iteration can be configured. In order to test what performance impact the packed version would have because

[6]**</dream·in·code>:** http://www.dreamincode.net/code/snippet4698.htm
[7]**Solvium:** http://www.solvium.de/programmierung/java/primfaktorzerlegung/

of the packed method's invocation via reflection, two more methods have artificially been introduced into the code, which would be called for every iteration. The more iterations the program uses to calculate π, the more precise π will be, but it will take more time for the program to finish.

PrimFac

The algorithm for PrimFac is very trivial and far from efficient, however it is sufficient for our purposes. In a loop a counter starting at 2 is incremented by 1 for each round. If an integer division is possible with the number to be factorized then the counter resembles a prime factor. In this case the counter is reset to 2 and the loop continues with the quotient until the counter grows greater than the quotient and the last prime factor has been found.

EncDec

EncDec creates 1,000 AES-256 keys, takes a test-string and successively encrypts it in a loop with all keys. Then it decrypts the 1,000 cipher-strings again and displays the clear-text. This entire process (consisting of key creation, encryption and decryption) is done ten times in a row.

4.2.2 Test Results

We tested an unpacked and a packed version of PiCalc, each with 400,000,000, 40,000,000 and 1,600,000,000 iterations. The packed version took about 5 times longer to execute, while there was a slight decrease in performance with the more iterations the program used. This supports our assumption that a packed program executes slower than an unpacked program.

For PrimeFactors again we measured the execution time when the program was packed and when it was unpacked. This time on average the packed program took a little less than twice as long to execute. In order to demonstrate that the loss of performance is indeed caused by method invocation via reflection we packed the whole program again, except for the core method which is called for every iteration of the central loop. This causes the program to call only one packed method for every execution. The unexpected and interesting result not only demonstrates that indeed the bottleneck is the calls for packed methods but it also shows that the packed program actually executes faster than the unpacked version, reducing execution time by about 5%.

By executing EncDec both observations from PrimeFactors could be confirmed: by calling only about 2,000 packed methods per encrypting/decrypting routine, the overhead for calling those methods still does not have a big impact on the overall execution speed. Again, however a considerable increase in execution speed for the packed version of EncDec—12%—could be observed.

A closer look at the execution time of certain parts of the EncDec helps to understand better why a packed program may run faster than an unpacked program without any overhead. When the en-/decryption routine is called for the first time in an unpacked program, it takes the program about three times longer to execute this routine than for the subsequent 9 runs. This causes the overall execution time to go up for several milliseconds. If the program, however, is packed, then the first run takes just as long as the following runs.

We could not explain this phenomenon as the answer probably lies in the concrete implementation of the virtual machine. The observation that the first round is slower than the following suggests that caching optimizations could be responsible. Unfortunately we could not clarify the exact reason why in some cases a packed program runs faster than an unpacked one.

To sum up we can say that the packing process has a minor impact on the performance, unless the packed methods are called many times during the execution of the program. In this case the expensive mechanism for calling a method via reflection has a noticeably negative effect on the performance. In some cases we observed that the packed program takes even less time than the unpacked program, probably due to caching optimizations.

4.3 Evaluation of Functionality

An important property of PPJ is that the functionality of a packed program remains the same. The packer may add its own functionality but the original functionality may stay the same. By packing a large quantity of programs, executing them after they have been packed and by observing their behavior and output we can say that the packer principally does not alter their behavior. However, due to the added unpacking code we have observed a side effect which reduces the efficiency of some types of malware.

Due to its methodological and practical nature, the stated approach for verifying the correct functionality seems to be unsatisfyingly vague. It is commonly known that software programs show errors, thus the software industry has to fight a similar problem in order to verify the correctness of a program according to its requirements. In fact this is done by applying *unit tests* which pursue this exact imperfect approach to increase reliability, trying to find the optimum between cost and gain. In most cases unit testing yields satisfactory results by finding the most relevant errors, however it cannot guarantee that all errors have been found before the software is published. Due to this it is common practice to provide *patches* to remedy errors which have not been discovered during development. When it comes to software that is written to support, save or sustain vital functions like human life or world-wide economics, errors cannot be afforded. Keeping in mind that, in essence, every state of a program is determinable, a computer program can be described with mathematical measures. By using mathematical logic, also the requirements can be cast into a mathematical form which can be used to express a formal proof which may or may not confirm the program's correct course of action. This discipline is often called *formal methods.*

In [13] Emerson describes *model checking* as an instance of the verification problem.

> "The verification problem is: Given program M and specification h determine whether or not the behavior of M meets the specification h. [...] The model checking problem is an instance of the verification problem. Model checking

Performance Test	Configuration	Method calls (packed)	Avg. exec. time unpacked (ms)	Avg. exec. time packed (ms)	Performance impact
	Iterations (accuracy)				
Calculation of π	400,000,000	400,000,001	5,714.00	28,019.00	490%
	40,000,0 00	40,000,001	577.67	2,932.33	508%
	1,600,000,000	1,600,000,001	22,903.67	123,240.67	538%
	Factorized number				
Prime factorization	78,762,876,538,239	54,829,531	1,349.00	2,401.00	178%
	78,762,876,538,239	1	–	1,301.00	96%
	123,456,789,012,345	7,556,169	192.00	397.00	208%
	123,456,789,012,345	1	–	195.00	102%
	987,654,321,098,765	662,841,193	16,185.00	28,086.00	174%
	987,654,321,098,765	1	–	15,482.00	96%
	100,000,039	100,000,039	2,470.00	4,285.00	173%
	100,000,039	1	–	2,350.00	95%
	Iterations				
En-/decryption	1,000	2,003	590.10	522.40	88%

Table 4.3: Performance tests on three packed programs were conducted, whose results are shown in this table. All programs can be configured, whereas this configuration has an impact mostly on execution time. The first program calculates π, the second one factorizes numbers and the third one encrypts and respectively decrypts a string in configurable iterations. Note that because of the overhead the packer adds to a program, it principally takes longer to execute in packed form. This depends heavily on the number of called packed methods. A program which calls more packed methods takes longer to execute than the same program, which is configured to call fewer packed methods.

> provides an automated method for verifying concurrent (nominally) finite state systems that uses an efficient and flexible graph search, to determine whether or not the ongoing behavior described by a temporal property holds of the system's state graph. The method is algorithmic and often efficient because the system is finite state, despite reasoning about infinite behavior. If the answer is *yes* then the system meets its specification. If the answer is *no* then the system violates its specification[...]."

Based on modal and temporal logic, LTL[8] was invented and published by Pnueli in [30] or branching time logics like CTL[9] in [7] which provide "a high degree of expressiveness permitting the ready capture of a wide range of correctness properties of concurrent programs, and a great deal of flexibility", states Emerson in [13]. Nowadays the ongoing work of 25 years allows formal verification on a large scale with many systems in industrial productive environments being verified also as a matter of routine. For more refer to [13] as this paper gives a good overview on formal methods and provides literature for further reading.

Given the practical nature of this thesis applying formal methods to verify the correctness of PPJ would cause too much outlay in view of the benefit. Whereas still, it should be stressed that a scientific work normally shouldn't follow standards which are rooted in economically-driven approaches. However, we are content with being able to say that, with high probability, PPJ is going to maintain the exact original functionality of a given program during the packing process.

During our tests with the programs we used for the performance tests (section 4.2.1), we never noticed and different behavior between the packed and the unpacked programs. Further tests with more complex open source programs indicate problems with concurrent threads. Considering that PPJ is still just a prototype this problem seems to be negligible.

As this thesis concentrates on the nature of malware it is interesting to know if malware would also work as a packed version and if any side effects would occur. One malware, Java/Agent.HC, tries to access a PHP-script on a web server with the goal of downloading an executable file, which is presumably malicious. The URL it accesses reads: http://affstor.tk/new/load.php?f=1&e=5. Once it is downloaded the downloader tries to execute it. In order to test this executable safely on a system running Microsoft Windows XP we installed a local web server and set up a PHP-script of the same name that the downloader would expect. This script returns calc.exe, the well-known (and harmless) Windows calculator application. Then we added an entry in the hosts-file, which is located at %SystemRoot%\system32\drivers\etc\hosts in order to let the domain affstor.tk point to localhost, thus to our local web server. This allowed us to safely run the Java/Agent.HC binary without manipulating the malware's source code and without requiring network access. Both tests, one with the unpacked version of the downloader and the other with the packed version, were successful: in both cases the downloader showed the same behavior and executed calc.exe, while in the packed version it was not detected by any virus scanner.

[8]**L**inear **T**ime **L**ogic
[9]**C**omputation **T**ree **L**ogic

We tested another malware, named Exploit.OSX.Smid, which is part of the Metasploit package. This applet exploits a buffer overflow vulnerability in the getSoundBank method in older versions of Sun's implementation of the Java VM allowing an attacker to execute arbitrary code. In the case of an applet this not just means that for example a backdoor could be installed on the victim's system but it also means that a breakout of the sandbox in which the applet is normally run is possible. The sandbox restricts an (unsigned) applet from invoking potentially dangerous methods such as accessing files on the host or creating or listening for network connections. The functionality-tests for this exploit were successful basically, however some complications arose for the packed version of the exploit.

The packer adds a custom class loader in order to be able to dynamically load the externalized class into the Java VM after it has been unpacked. The problem is that an unsigned applet is not allowed to create and use its own class loader to prevent the applet from reloading any code during runtime. A way to prevent this is to sign the applet, which gives the applet all privileges of a 'normal' Java application. If the applet is signed with a self-signed certificate, which is easy to create, the browser is going to pop-up an information dialogue which informs the user about the certificate's invalidity, but still gives the user the opportunity to execute the applet anyway. Even though it is highly probable that the user is going to ignore this warning and let the applet proceed, and even if an attacker could somehow bring a CA to sign the malicious applet the main issue lies somewhere else. The problem is that by elevating the applet's permissions by signing it, which is mandatory in order for the packed applet to execute properly, it makes the exploit's purpose almost void. The exploit shows that a breakout from the sandbox is possible, but with the exploit having all privileges anyway a vulnerable VM is not necessary anymore to perform the exploit's malicious behavior. All that is left is the exploit's impact on the system, but the real trick—breaking the applet's sandbox—is no good anymore if the user grants the applet full access anyway.

Although the packer does not change the packed program's behavior it adds code which causes some complications for malicious applets. For classic Java programs be it malicious or harmless ones we could not find any problems regarding a change in functionality, except for problems with concurrent threads, as mentioned above.

5 Closing remarks

PPJ as a prototype still holds much room for improvement. For instance there are weaknesses according to the packing technique. Once a polymorphic packer, whose concept is based on ‘security by obscurity’, has been analyzed it should be fairly easy to develop an unpacker which essentially performs the same operations on the packed payload as the stub does. Then the normalized code can be used again and checked against the signatures in the virus database.

Primarily, it was planned to develop a packer that implements a similar technique like emulation obfuscators (cf. 2.4.7). The efficiency would have been much higher as such obfuscators operate in a more fine-grained way, on the level of statements instead of packing whole functions. Thus, they increase the level of complexity. However, Java’s rich security concepts, its architecture implementing deliberate restrictions and Java itself as a very high-level programming language would have made it too complex and too time-consuming to implement such a packer in line of this work. Yet, such packers have been developed using languages like C which allow the developer to make use of of pointers to access arbitrary memory frames or to manipulate low-level structures like registers in the CPU by incorporating assembly code. Mighty development tools like these give developers more control over the system than Java allows to. With Java the developer has a more abstract view on the system which allows a more comfortable development but with considerable loss of control. Although it should be possible to implement an emulation obfuscator in Java, due to the lack of adequate tools the development of such a packer is probably going to turn out quite intricate.

The question is if such a packer is not going to suffer from significant performance problems regarding execution time. As we have seen, a packed executable runs slower if many invocations via reflection have to be made. If the packing granularity is brought down from the level of methods to the level of single statements the amount of reflective invocations would increase significantly. So, probably another technique for dynamic code execution should be introduced when a packer of this kind is developed. A prime candidate for this could be Java’s *instrumentation API*. As the Java API specification [27] says: The Java instrumentation API “provides services needed to instrument Java programming language code. Instrumentation is the addition of byte-codes to methods for the purpose of gathering data to be utilized by tools. Since the changes are purely additive, these tools do not modify application state or behavior.”. Although originally intended to be used to assist debugging, logging or monitoring this mechanism probably could also be utilized to dynamically re-load needed code during runtime. A stub could act as an interceptor to fill a formally empty method (which has been linked during compile-time) with code before it is it called regularly without reflection. On the one hand this mechanism could maybe avoid the time-consuming calls via reflection and render custom class loaders unnecessary on the other hand. The latter would be a benefit for packed applets, cf. 4.3. However, all of the above are just assumptions, it still has to be verified wether in fact the instrumentation API performs as assumed.

Another downside of PPJ is its inability to pack fields. Often also they hold important information which is detected by malware scanners. It would be desirable to also be able to hide the fields' values, as well as the fields' names.

Also, a more effective parser would improve the packer's usability. Due to the incapability of Janino's parser to resolve some identifiers (cf. 3.4) some constructs have to be restated in order for the packer being able to process it.

As the packer primarily has been developed to test malware and malware is available mainly as binary without source code, probably it would also be better to develop a future packer to process class files instead Java source code.

In this thesis we concentrated on program obfuscation carried out by runtime-packing. On the one hand we gave an introduction into program obfuscation, highlighting several aspects of the topic, followed by several techniques of runtime-packing and possible counteractive measures. Over time many different techniques have been invented to protect executables for various purposes. For many of them successful countermeasures have been developed, too, for others efficient antidotes are still to be found.

Furthermore, in line of this work a novel java-based polymorphic runtime-packer was developed with the goal to obfuscate Java-malware and to explore the possibilities in order to understand the challenges that malware-analysts have might have to face in the future. The packer's concept and implementation details were described followed by evaluation details. It could be shown that it is possible to develop a polymorphic packer for Java programs that implements an quite effective technique to thwart state-of-the-art and up-to-date A/V software packages. Except for some malware binaries mostly all of them couldn't be detected anymore after packing. The performance in respect to execution time or reliability (thus maintained functionality) was not significantly affected by the packing process.

Originally for the practical part of this thesis it was planned to do an evaluation of state-of-the-art A/V software. The performance in relation to malware that is obfuscated with packers implementing modern obfuscation techniques should have been evaluated. However, finally is was decided to discard this topic, as there are too many obstacles in the way to perform this evaluation properly. The intransparency in relation to the closed nature of A/V software, as well as the intransparency regarding the diversity of malware which is difficult to find samples for in large amounts and which is hard to normalize as it primarily occurs already packed in the wild would have made it hard and elaborate to perform this evaluation appropriately. So, it was decided to turn towards implementing a PPJ.

Looking back we can say that we brought our project to a successful end. We managed to build a packer which is able to conceal Java-based malware so that modern static malware-scanners cannot detect it anymore, except for a small percentage in a few cases. Furthermore all other requirements could be satisfied. Though we have achieved good evaluation results regarding the degree of obfuscation, there is no doubt that sooner or later also unpacking algorithms would be found, implemented and incorporated into

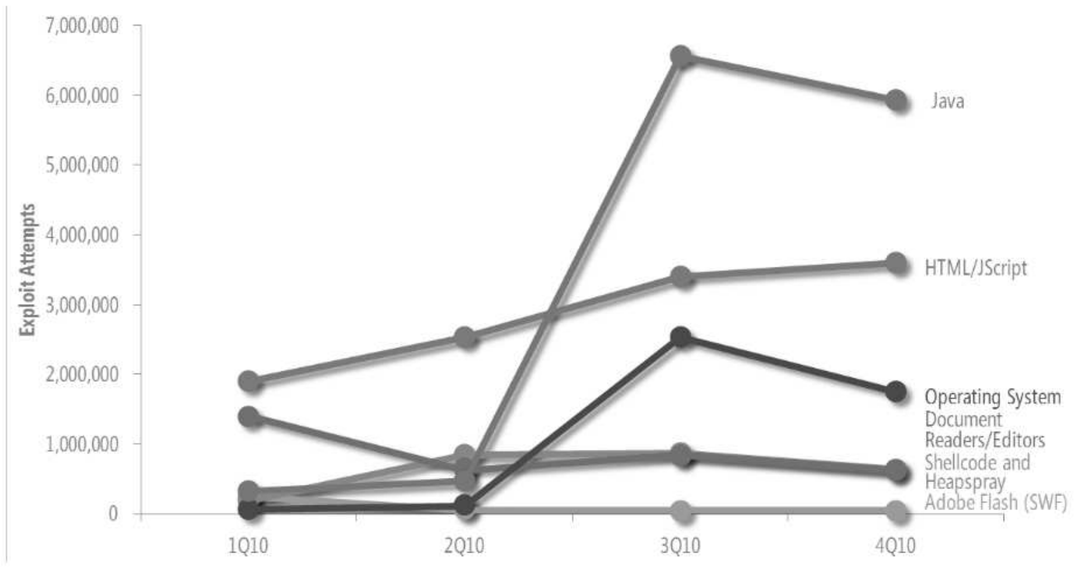


Figure 5.1: The graph, taken from Microsoft's Security Intelligence Report [6], shows that Java-based exploits have significantly increased since 2010.

malware scanning engines if packers similar to PPJ would come into circulation. It is no unfounded consideration that packer for Java may become more interesting for malware authors in the future as Java-based malware seems to be in the ascendant, recently. Microsoft's Security Intelligence Report Volume 10 [6] which was published in the second quarter 2011 states that in "3Q10, the number of Java attacks increased to fourteen times the number of attacks recorded in 2Q10, driven mostly by the exploitation of a pair of vulnerabilities in versions of the Sun (now Oracle) JVM, CVE-2008-5353 and CVE-2009-3867. Together, these two vulnerabilities accounted for 85 percent of the Java exploits detected in the second half of 2010". Obviously Java is becoming a popular target for exploits which is supported by fig. 5.1 showing the increase of Java-based exploits. So A/V manufacturers probably have to be prepared for stronger headwinds which they might encounter on this sector.

List of Figures

List of Tables

Bibliography

[1] Aho, A.V., M.S. Lam, R. Sethi, and J.D. Ullman: *Compilers - Principles, Techniques & Tools.* Pearson Education, Inc., 75 Arlington Street, Suite 300, Boston, MA 02116, second edition ed., 2007.

[2] Barak, B.: *Can we obfuscate programs?* http://www.math.ias.edu/~boaz/Papers/obf_informal.html, visited on 11/04/09.

[3] Barak, B., O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.P. Vadhan, and K. Yang: *On the (im)possibility of obfuscating programs.* In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pp. 1–18, London, UK, 2001. Springer-Verlag.

[4] Beale, J.: *"security through obscurity" ain't what they think.* http://web.archive.org/web/20010813093230/http://securityportal.com/beale/beale20010720.html, visited on 11/06/09.

[5] Borwein, J., D. Bailey, and R. Girgensohn: *Experimentation in Mathematics - Computational Paths to Discovery.* A K Peters, 2003.

[6] Cavit, D., M. Meyer, and J. Salido: *Microsoft security intelligence report.* Tech. Rep. 10, Microsoft Corporation, 2011.

[7] Clarke, E.M. and E.A. Emerson: *Design and synthesis of synchronization skeletons using branching time temporal logic.* In *Logic of Programs, Workshop*, pp. 52–71, London, UK, 1982. Springer-Verlag.

[8] Collberg, C. and J. Nagra: *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 1st edition ed., 2009.

[9] Collberg, C., C. Thomborson, and D. Low: *A taxonomy of obfuscating transformations.* Techn. rep., The University of Auckland, New Zealand, 1997.

[10] Collberg, C., C. Thomborson, and D. Low: *Manufacturing cheap, resilient, and stealthy opaque constructs.* In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pp. 184–196, New York, NY, USA, 1998. ACM.

[11] DataRescue, 40 Bld Piercot, 4000 Liege, Belgium: *Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables.*, 2005. http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf, visited on 03/18/10.

[12] Eilam, E.: *Reversing: Secrets of Reverse Engineering.* Wiley Publishing, Inc., 10475 Crosspoint Boulevard, Indianapolis, IN 46256, 2005.

[13] Emerson, E.A.: *The beginning of model checking: A personal perspective.* http://www.model.in.tum.de/um/25/pdf/Emerson.pdf, visited on 05/06/11.

[14] Ferrie, P.: *Attacks on more virtual machine emulators.* Symantec Security Response, December 2006.

[15] Ferrie, P.: *Attacks on virtual machine emulators.* Symantec Advanced Threat Research, 2006.

[16] Ferrie, P.: *Anti-unpacker tricks.* CARO Workshop, May 2008. http://pferrie.tripod.com/papers/unpackers.pdf, visited on 01/10/10.

[17] Grover, S.: *Linkers and loaders.* http://www.linuxjournal.com/article/6463, visited on 12/06/09.

[18] Harrison, W.A. and K.I. Magel: *A complexity measure based on nesting level.* SIGPLAN Not., 16(3):63–74, 1981, ISSN 0362-1340.

[19] Kang, M.G., P. Poosankam, and H. Yin: *Renovo: a hidden code extractor for packed executables.* In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pp. 46–53, New York, NY, USA, 2007. ACM.

[20] Levine, J.R.: *Linkers and Loaders.* Mogran Kaufmann Publishers Inc., 1st edition ed., October 1999.

[21] Lyda, R. and J. Hamrock: *Using entropy analysis to find encrypted and packed malware.* IEEE Security and Privacy, 5:40–45, 2007.

[22] Mallach, E.G.: *On the relationship between virtual machines and emulators.* In *Proceedings of the workshop on virtual computer systems*, pp. 117–126, New York, NY, USA, 1973. ACM.

[23] Martignoni, L., M. Christodorescu, and S. Jha: *Omniunpack: Fast, generic, and safe unpacking of malware.* In *ACSAC '07: Proceedings of the 23rd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*, pp. 431–441, 2007.

[24] McCabe, T.J.: *A complexity measure.* In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, p. 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[25] Microsoft Corporation: *Microsoft Portable Executable and Common Object File Format Specification*, 2008. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, visited on 12/09/09.

[26] Oracle: *Understanding Extension Class Loading*, 1995. http://download.oracle.com/javase/tutorial/ext/basics/load.html, visited on 04/21/11.

[27] Oracle: *Java™ Platform, Standard Edition 6 API Specification*, 2011. http://download.oracle.com/javase/6/docs/api/, visited on 05/08/11.

[28] Perdisci, R., A. Lanzi, and W. Lee: *Classification of packed executables for accurate computer virus detection.* Pattern Recognition Letters, 29:1941–1946, October 2008.

[29] Pietrek, M.: *Peering inside the pe: A tour of the win32 portable executable file format.* Microsoft Systems Journal, 1994.

[30] Pnueli, A.: *The temporal logic of programs.* In *18th Annual Symposium on Foundations of Computer Science (FOCS 1977)*, pp. 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[31] Popek, G.J. and R.P. Goldberg: *Formal requirements for virtualizable third generation architectures.* Commun. ACM, 17(7):412–421, 1974, ISSN 0001-0782.

[32] Rolles, R.: *Unpacking virtualization obfuscators.* WOOT '09, August 2009. http://www.usenix.org/event/woot09/tech/full_papers/rolles.pdf.

[33] Royal, P., M. Halpin, D. Dagon, R. Edmonds, and W. Lee: *Polyunpack: Automating the hidden-code extraction of unpack-executing malware.* In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pp. 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[34] Scherzo: *Inside code virtualizer.* CodeBreakers Journal, 4(2), 2007.

[35] Schneier, B.: *Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C.* John Wiley & Sons, Inc., 1996.

[36] Schroder, C.: *Security through obscurity? it's not all bad*, 2007. http://www.enterprisenetworkingplanet.com/netsecur/article.php/3680286, visited on 11/11/09.

[37] Sharif, M., A. Lanzi, J. Giffin, and W. Lee: *Automatic reverse engineering of malware emulators.* In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pp. 94–109, Washington, DC, USA, 2009. IEEE Computer Society.

[38] Sosnoski, D.: *Java programming dynamics, Part 1: Java classes and class loading.* IBM, 2003. http://www.ibm.com/developerworks/java/library/j-dyn0429/, visited on 04/21/11.

[39] Ször, P. and P. Ferrie: *Hunting for metamorphic.* In *In Virus Bulletin Conference*, pp. 123–144, September 2001.

[40] Tanenbaum, A.S.: *Modern Operating Systems.* Prentice Hall, 2nd edition ed., 2001.

[41] Unkrig, A.: *Janino.* Codehaus, 2010. http://janino.net/, visited on 04/26/11.

[42] Westley, B.: *westley.c.* 5th International Obfuscated C Code Contest, 1988. http://www.ioccc.org/1988/westley.c.

[43] Yan, W., Z. Zhang, and N. Ansari: *Revealing packed malware.* Security & Privacy, IEEE, 6(5):65–69, 2008.

[44] Yason, M.V.: *The art of unpacking.* In *Black Hat Briefings USA 2007*, 2007.