Automatic Malware Signature Generation

Karin Ask

October 16, 2006

Abstract

The times when malware researchers could spend weeks analyzing a new piece of malware are long gone. Today new malicious programs are written and distributed at such speed that it just is not possible. Virus scanners are the most common countermeasure against malware attacks and they need up-to-date signatures to successfully identify malware. This thesis describes Autosig, a program for automatic generation of malware signatures.

The generation of signatures is based on the fact that most malware come in many different variants, but still share some invariant code. Using statistical data on how often certain byte combinations appear in legitimate files, Autosig extracts a substring from this invariant code to generate a signature. The signatures are tested and those that fail to pass all tests are discarded. By remembering all discarded signatures, Autosig learns which code to avoid.

This technique has turned out to be successful in many of the time consuming routine cases, leaving the human analysts more time to generate working signatures for more complicated malware. It has also turned out helpful in replacing overlapping and redundant signatures, leading to a smaller signature database.

Acknowledgements

This work has been carried out at Ikarus Software GmbH and I want to thank them for giving me the opportunity to work on such an interesting subject.

I also want to thank Christopher Krügel and Engin Kirda at TU Wien for getting me interested in Internet security in the first place and for their help and advice throughout my work on this thesis.

Christian Schulte at KTH Stockholm has been very helpful in giving me feedback and valuable comments on my writing.

Moreover my parents Ulla & Gösta Ask deserve a big thank you for their constant help and support, not just during the writing of this thesis but throughout my entire studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1984 Dr. Frederick B. Cohen defined a major computer security problem that he called a virus - a program able to attach itself to other programs and cause them to become viruses as well [10]. Today, twenty-two years later, the problem has grown more severe than anyone could imagine. The popularity of the Internet, a lack of diversity in the software running on Internet-attached hosts and an increasing variety of exploitable vulnerabilities in software has brought with it an explosion in the number of malicious software attacks.

The most common countermeasure to these attacks are virus scanners, that search files for *signatures*, small bytepatterns that correspond to malicious code. Since only already known malware, where a signature is available, can be detected this approach requires a fast response to new malware. As long as there is no signature for a certain malware it will pass through the virus scanner as a non-malware file. The signatures have to be carefully chosen, so that they are not found in innocent files by coincidence. There is always a trade-off between fast generation of signatures and avoiding false positives.

In the best case scenario, every new malware strain is analyzed and a signature corresponding to its unique function is generated. The problem is that this type of analysis takes time. In January 2006 2,312 new samples of malware were spotted [28]. Even if this is a new record for the number of malware being spotted in one month, it is not hard to see that the analysts in an anti-virus lab do not have enough time to analyze new malware and generate a working signature for it. Malware writers are well aware of this weakness and have challenged anti-virus software with zero-day, short-span and serial-variant attacks as well as other new propagation techniques.

Motivated by the slow pace of manual signature generation, the goal for this master thesis has been to develop and implement the tool Autosig, that generates signatures *automatically*. Autosig identifies invariant code among several instances of the same malware strain and uses statistical data on this code to generate signatures. The relevance of such an approach in automatically generating signatures has already been shown by Kephart and William [19].

Much malware today are generated with the help of virus generation kits, or are slightly modified copies of already known malware. In practice we find that these malware instances often contain some invariant code. By identifying this code it is possible to generate a signature that finds all variants of the same malware. Such generic signatures help reduce the memory needed to store signatures and speed up the time needed to scan files.

It is not anticipated that working signatures can be generated automatically for all malware. Much malware evade detection by polymorphism techniques, through which a program may encode and re-encode itself and thus changing its payload between every attack. Even though there is invariant code in these files too, such as the routine for encoding, this type of malware normally need to be analyzed by human experts. However, by generating some signatures automatically, the human analysts will have more time for these complicated attacks.

1.1 Goals

This Master Thesis analyzes the existing signature generation workflow at Ikarus Software GmbH, an Austrian anti-virus company, with its benefits and drawbacks. After this analysis the tool Autosig that automatically generates signatures is to be developed. The main question to be answered by this work is whether automatic signature generation is feasible. When yes, where can the main profits be found? Is it possible to enhance the quality of the signatures generated? Can a faster reaction to new malware be achieved this way?

The goals can be summarized as follows:

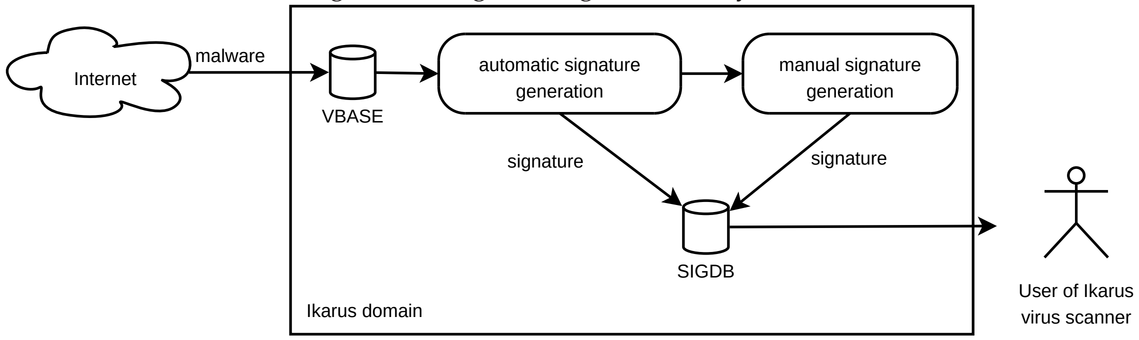
- Analyze the existing workflow used by the Ikarus Analysis team, how to manually generate malware signatures.
- Develop a concept how to automate the process of generating signatures for malware samples.
- Develop a working prototype of Autosig to demonstrate how to automatically generate signatures from incoming malware samples.
- Autosig should try to generate signatures for all malware samples, independent of file type or target machine.

The ultimate goal is then to integrate Autosig with the other tools at Ikarus so that signatures are generated without any human intervention, when possible. An overview of such a system is schematically shown in figure 1.1.

1.2 Structure of this thesis

This first chapter introduces the field and gives a motivation to why this work is needed. It presents the problem and draws an outline of the proposed solution. The second

Figure 1.1: Signature generation system


Internet
malware
VBASE
automatic signature generation
manual signature generation
signature
signature
SIGDB
Ikarus domain
User of Ikarus virus scanner

SIGDB is the database for storing working signatures. All malware samples are stored in the database VBASE.

chapter gives an overview of malware in general and how to deal with malware attacks. It also compares the anti-virus solution at Ikarus Software to these. In the third chapter signatures are discussed in detail and the present signature generation workflow at Ikarus is analyzed. The design and implementation of Autosig is explained in the fourth and fifth chapter. In the sixth chapter some test results are presented. The seventh and last chapter concludes the thesis and outlines future work. In the appendixes log files from running Autosig can be found.

The term ″virus″ is often used to describe all kinds of malicious software, including those that are more properly classified as worms, trojans or something else. In this thesis the term malware will be used instead, to avoid any misunderstandings.

1.3 Glossary

In this section, important technical terms that are used throughout this thesis are described.

Polymorphic virus A polymorphic virus is a virus that changes its appearance in host programs. For instance, it encrypts its body with a different key each time, and prepends a decryption routine to itself. The decryption routine is then mutated randomly across virus instances. This way the virus can avoid detection by signature-based virus scanners.

PE format PE stands for Portable Executable. An executable file is a program in binary code that can be run without any further human intervention. Windows Portable Executable files are simply executable files that work across all Microsoft 32-bit operating systems, which is why the majority of malware for Windows written today is written in this format. Since most malware is targeted at Windows, PE files are generally the dominant malware format.

Honeypots In computer terminology, a honeypot is a trap set to detect, deflect or in some manner counteract attempts at unauthorized use of information systems. Generally it consists of a computer, data or a network site that appears to be part of a network but which is actually isolated and protected, and which seems to contain information or a resource that would be of value to attackers [36].

Reverse Engineering Reverse engineering is the process of analyzing an object to create a representation of it at a higher level of abstraction [9]. The process of reverse engineering an executable program typically begins with disassembly, which translates machine code to assembly code. This is then followed by various decompilation steps that aim to recover the source code in some high-level language from the assembly code.

Reverse engineering is often deterred by obfuscation and it is often very hard to completely reverse engineer a program.

Chapter 2

Malware

The word malware is a portmanteau of ”malicious software” and means software designed to infiltrate or damage a computer system, without the owner's consent [36]. Malware is a collective name for viruses, worms, trojan horses, spyware, adware and their likes. Commonly the word ”virus” is used to describe all the above, even though actual viruses make out a pretty small number of the existing malware. In 2004 viruses only stood for about 4 percent of all malware, whereas worms stood for the majority [23].

The following chapter gives an overview of malware in general. It also addresses some different ways to detect malware, how to approach malware attacks and describes the anti-virus solution at Ikarus.

2.1 A Brief History

In the year 1972, the first distributed computing virus was unleashed upon the ARPANET under the name Creeper. The virus did not perform any malicious actions such as modifying the file system, nor did it even remain on a single computer for any lengthy amount of time. Written for the then popular operating system Tenex, the virus utilized a system's modem to spread itself to other machines connected to the ARPANET. Once the virus successfully transferred itself to a remote machine, it would delete itself from the previous host machine. Infection was made evident by the following message being displayed on the screen:

```
"I'M THE CREEPER: CATCH ME IF YOU CAN"
```

A subsequent development was The Reaper, a ”search and destroy” program which stalked Creeper across the same network, eliminating it from all its hiding places. It has been postulated that The Reaper was thus the first anti-virus program [33].

One of the earliest verified reports of a computer virus was issued in 1981. Elk Cloner [15] infected the boot sector disks on Apple II PCs and copied itself to the boot sectors of diskettes as they were accessed. On every 50th boot, the Elk Cloner hooked the reset handler; thus, only pressing reset triggered the payload of the virus. The virus contained a number of payloads: it inverted the screen, caused text to flash and displayed the following poem:

```
Elk Cloner:  The program with a personality

        It will get on all your disks
          It will infiltrate your chips
            Yes it’s Cloner!

        It will stick to you like glue
          It will modify ram too
            Send in the Cloner!
```

In 1984 Dr. Frederick B. Cohen introduced the term virus based on the recommendation of his advisor Professor Leonard Adleman, who picked the name from science fiction novels. Cohen also provided a formal mathematical model for computer viruses [11].

Reports of the Brain-virus started to appear in newspapers worldwide in 1986. This boot sector virus, which infected 360K floppy disks, quickly spread worldwide with reports being received from Asia, the Middle East, Europe and the US. The virus had been written by two pakistani brothers who ran a software company and were reportedly annoyed at the extent of software piracy and theft taking place in their country. The virus was an attempt to define the extent of such copying.

One of the first mass-mailers was the infamous CHRISTMA EXEC worm [6]. In December 1987 it caused the first major network attack. The CHRISTMA EXEC was a chain letter that spread from the University of Clausthal Zellerfeld in Germany onto Bitnet, via a gateway to the European Academic Research Network (EARN), and then onto IBMs internal network known as VNet. The CHRISTMA EXEC was written in the script language REXX introduced by IBM and spread on VM/CMS installations. It relied on social engineering and since users were happy to follow the instructions in the script it was able to spread rapidly. When run, it drew a Christmas tree on the screen, looked around for user IDs on the system and sent a copy of itself to these, using the SENDFILE command.

The Morris Internet worm [29] was released on 2 November 1988. The worm replicated by exploiting a number of bugs in the Unix operating system on VAX and Sun Microsystems hardware, including a bug in sendmail (an electronic mailing program) and in fingerd (a program for obtaining details of who is logged on to the system). Stanford University, MIT, the University of Maryland and Berkeley University in California were infected within five hours of the worm being released. The NASA Research Institute at Ames and the Lawrence Livermore national laboratory were also infected. It is believed that about 6,000 computer systems were infected, about 10 percent of the Internet at that time. The worm consisted of some 4,000 lines of C code and once it was analyzed, specialists distributed bug fixes to sendmail and fingerd which prevented the worm from propagating further.

The early ´90s saw the first polymorphic computer viruses and the Mutation Engine (MtE), written by the infamous bulgarian virus writer Dark Avenger, appeared. The first version of the MtE was released in the summer of 1991. It was an object that could be linked to any simple virus. Writing polymorphic viruses was difficult for novice virus writers. The concept of the MtE is that by calling the mutation engine function a polymorphic shell is built around the simple virus inside it.

One of the first viral code construction set for IBM PC viruses was the Virus Creation Laboratory (VCL) that was released in the early nineties. This set allows the user to generate viruses with the help of a menu system where she can choose virus type, objects to infect (COM or/and EXE), presence or absence of self encryption, measures of protection from debugging and much more. After VCL came many other virus generation kits, such as Phalcon-Skism Mass-Produced Code Generator (PS-MPC) and Next Generation Virus Creation Kit (NGVCK). Such generators can be used by just about anyone, which opened the world of malware writing to kids, unfamiliar with Assembly language or polymorphism. Probably the most famous result of such virus generation kits is the Anna Kournikova worm, created by a 20-year-old Dutchman in 2001, using the VBS Worm Generator. It was sent via email and had an attachment that appeared to be a graphic image of Russian tennis star Anna Kournikova. However, the attachment was really a Visual Basic Script that sent the worm on to all addresses listed in the Microsoft Outlook email address book. The worm also uses encryption to hide itself, a feature included in the virus generation kit.

By the end of the 1990s, malware were capable of attacking the functioning of computer hardware, with Chernobyl being the first virus to affect the BIOS of a computer.

The development of the Internet opened up the whole world to malware writers around 1995, so that malware were no longer constrained to spreading simply via the exchange of floppy disks. Together with the development of worms malware could propagate rapidly via email and the web, with the result that outbreaks often became global phenomena. Table 2.1 shows the development in malware numbers over the last ten years [33]. The top 5 malware of the current year is presented together with the number of found instances. The increase is significant.

Table 2.1: Malware prevalence 1995 & 2005.

Year 1995		Year 2005	
Malware Name	Instances	Malware Name	Instances
Form	255	W32/Sober	16564070
AntiEXE.A	203	W32/Netsky	607855
Parity_Boot	167	W32/Mytob	53800
Empire.Monkey.B	138	W32/Bagle	97659
AntiCMOS	116	W32/Mydoom	79404

2.2 Classification of Malware

The need to classify malicious programs according to a unified nomenclature is about as old as computer viruses themselves. Obviously this is not an easy work with many classes overlapping or being closely related. In some cases researchers also disagree on the classification, even within CARO (Computer Antivirus Researchers Organization) [7]. For example whether worms should be classified as a subclass of viruses or not has given rise to many discussions. Some of the most common malware classifications, supported by a majority, are listed in the following.

Viruses As defined earlier, a computer virus is code that recursively replicates a possibly evolved copy of itself. Viruses infect a host file or system area, or they simply modify a reference to such objects to take control and then multiply again to form new generations.

Worms are typically standalone applications without a host program. They primarily replicate on networks, usually without any help from a user. However, some worms also spread as a file-infector virus and infect host programs, which is why they are often classified as a special subclass of viruses. If the primary target of a virus is the network, it should be classified as a worm [31].

Trojan Horses (trojans) portray themselves as something other than what they are at the point of execution. Though it may advertise its activity after launching, this information is not apparent to the user beforehand. A trojan neither replicates nor copies itself, but causes damage or compromises the security of the computer. A trojan must be sent by someone or carried by another program and may arrive in the form of a joke program or software of some sort. The malicious functionality of a trojan may be anything undesirable for a computer user, including data destruction or compromising a system by providing a means for another computer to gain access, thus bypassing normal access controls [30].

For example, on UNIX-based systems, intruders often leave a modified version of ”ps” (a tool to display a process list) to hide a particular process ID (PID).

Probably the most famous trojan horse is the AIDS TROJAN DISK [1] that was sent to about 7 000 research organizations on a diskette. When the trojan was introduced to the system, it scrambled the names of files and filled the empty areas of the disk. A recovery solution was offered in exchange of a bounty.

Exploits are programs or techniques that take advantage of a vulnerability in software. Exploits can be used for breaking security, or otherwise attacking hosts over the network. Its goal is to run a program on a system automatically or provide some other form of more highly privileged access to the target system.

Penetration testers also create a form of exploit code. Depending on the actual use of the exploit the exploitation might be malicious in some cases but harmless in other - the severity of the threat depends on the intention of the attacker.

Spyware are programs that have the ability to scan systems or monitor activity and pass this information on to the attacker. Common information that may be actively or passively gathered is passwords, log-in details, account numbers, personal information, individual files or other personal documents. Spyware may also gather and distribute information related to the user's computer, applications running on the computer, Internet browser usage or other computing habits.

Spyware frequently attempts to remain unnoticed, either by actively hiding or by simply not making its presence on a system known to the user. Spyware can be downloaded from Web sites (typically in shareware or freeware), email messages, and instant messengers. Additionally, a user may unknowingly receive and/or trigger spyware, for example by accepting an End User License Agreement from a software program linked to the spyware [30].

Downloaders are programs that install a set of other items on the machine under attack. Usually a downloader is sent in email, and when it is executed, it downloads malicious content from a website or some other location and then extracts and runs its content.

Keyloggers are applications that monitor a user's keystrokes and send this information back to the attacker. This can happen via email or to a server controlled by the attacker. Information commonly collected is email and online banking usernames, passwords, PINs and credit card numbers. Attackers often use keyloggers to commit identity theft.

Keyloggers are also sold as legitimate applications, often as monitoring tools for concerned parents (or suspicious spouses).

Rootkits are components that use stealth to maintain a persistent and undetectable presence on the machine. Actions performed by a rootkit, such as installation and any form of code execution, are done without end user consent or knowledge.

Rootkits do not infect machines by themselves like viruses or worms, but rather, seek to provide an undetectable environment for malicious code to execute. Attackers typically leverage vulnerabilities in the target machine, or use social engineering techniques, to manually install rootkits. In some cases rootkits can be installed automatically upon execution of a virus or worm or simply by browsing a malicious website [30].

Once installed, an attacker can perform virtually any function on the system to include remote access, eavesdropping, as well as hide processes, files, registry keys and communication channels. Rootkits exists for most operating systems.

Hoaxes or Chain letters usually arrive in the form of an email, warning about a new computer virus infection and ask the recipient of the message to forward it to others. End users then spread the email hoax to others, "replicating" the message on the Internet by themselves and overloading email systems.

Combining two or more of the categories above can lead to even more powerful attacks. For example, a worm can contain a payload that installs a backdoor to allow remote access. When the worm replicates to a new system (via email or other means), the backdoor is installed on that system, thus providing an attacker with a quick and easy way to gain access to a large set of hosts.

In The Wild

Another classification of malware, often used by malware researchers and anti-virus companies is whether the malware has been seen *in the wild* or not.

In The Wild Viruses The term in the wild was coined by the IBM researcher Dave Chess, who used it to describe computer viruses encountered on production systems [17]. The organization Wildlist [37] keeps a list of current malware in the wild and anti-virus companies are assumed to find 100% of these. Today the definition of the term is typically a malware that has been seen by at least two independent wildlist submitters in at least two different regions.

Zoo Viruses Zoo Viruses are the opposite of in the wild viruses. It is a malware which is rarely reported anywhere in the world, but which exists in the collections of researchers. Its prevalence can increase to the point that it is considered to be in the wild. Though zoo viruses seldom pose a threat to the common user, they might prove some concept, making the detection of zoo viruses important for anti-virus software.

2.2.1 Malware Environments

Over the years, malware have appeared on many hardware platforms, including Apple II, C64, Atari ST, Amiga, PC and Macintosh. Mainframe systems and handheld systems such as PalmPilot, Symbian phones and the Pocket PC have also been attacked. However, the largest set of computer malware exist on the IBM PC and its clones [31].

Most malware spread in executable binary form, which typically is dependent on the underlying computer architecture. This explains why a virus for an Apple II is generally unable to infect an IBM PC and vice versa. In theory it would be feasible to create a multi-architecture malware, but because of the particularities of the underlying architecture it is no easy task. It can be hard for malware to propagate on machines with other CPUs than the one for which it is created. This is because the same instruction has different opcodes on different CPUs. A NOP (no operation) is for example defined `0x90` on Intel CPUs, but on VAX it is `0x01`. Malware are also often dependent on operating systems and their different versions, file systems or interpreted environments, such as the macro environment in Microsoft Office Products.

In march 2001, the PeElf proved that it is possible to create a cross-platform binary virus. It is able to infect both Windows PE and Linux ELF executables. PeElf is a proof of concept virus and has not been seen in the wild.

2.3 Malware Naming

Naming malware is a challenging task. While it was still humanly possible (when the number of known malware did not exceed 10 000), a committee met regularly at anti-virus conferences and decided what the name of each known malware should be [2]. Today anti-virus companies and researchers receive hundreds, sometimes thousands or more, new malware samples each month and the problem of naming malware by some common name is getting hard, if not impossible. There is not enough time to agree on a common name for each malware in the wild before responding to the threat.

Though assuring that each malware has the same name across all anti-virus vendors is impossible, some attempts have been made to make sure that the names comply to certain standards. In 1991 the founding members of CARO (Computer Antivirus Researchers Organization) designed a computer malware naming scheme for use in anti-virus products [3]. In 2002 this standard was expanded [16] and it is widely accepted as the basis for malware naming. The following is a short outline of this standard.

The general format of a Full CARO Malware Name is:

```
[<type>://][<platform>/]<family>[.<group>]
          [.<length>].<variant>[<modifiers>][!<comment>]
```

In practice very few malware requires all name components. Practically anything other than the family name is an optional field. The most commonly used identifiers are explained in the following.

The malware type-identifier specifies whether the malware in question is a virus, a trojan, a dropper, an intended virus, a malware generator kit or garbage. The list of possible types does not contain the type worm.

The platform-specifier denotes the operating system, interpreter or other kind of runtime environment necessary for the malware to function.

The family name is the most important component, given that it is the only one that must be reported to users. Thus, the most confusion is caused if there is inter-vendor disagreement over this component of a malware's name. Malware is grouped into families based on code similarity.

Group names have been retained mainly for backwards compatibility - many old DOS virus families are further divided into groups. The use of group names is strongly discouraged when naming new malware families.

An important issue to bear in mind when discussing malware names is the distinction between the formal, technically correct names used when anti-virus researchers are talking among themselves, and the simpler naming forms that are required when reporting a malware detection to an end user. For example, while technically correct the name "virus://{W97M,X97M,PP97M}/Tristate.A" should not be presented to end users. When found in a file, a better choice is probably to state that "Tristate.A" has been detected.

2.4 Detecting Malware

Over the years malware writers have been very ingenious in evolving their creations. Malware have evolved with regard to replication and spreading mechanisms, as well as techniques used to prevent analysis and/or detection. Such techniques include anti-debugging, encryption, using exe-packers, entry point obscuring etc. Even though it has been proved that there is no algorithm that can perfectly detect all future viruses in finite time [8, 10], malware researchers and anti-virus companies have been just as creative. Today there are many different ways to detect malware and protect computer users against malware attacks. In fact, the reality is that without anti-virus programs the Internet would be brought to a standstill because of the traffic undetected malware would generate.

Virus scanners are the most popular form of current anti-virus software and has been so for a long time. Other, generic, solutions such as behavior blockers or integrity checkers are also widely adopted, but scanners are the market´s accepted solution, regardless of it drawbacks. Ikarus Software GmbH uses a virus scanner to detect malware.

It is important to note that not all techniques can be applied to all malware and this should also not be required. Just because one technique cannot be used all the time does not mean it is completely ineffective. It is enough to have many different techniques, one of which will be a good solution to block, detect or disinfect a particular malware.

Most anti-virus product have some function to disinfect files or in other ways deal with detected malware. This is not the focus of this Master Thesis. Instead some available methods for detecting malware are described in the following.

2.4.1 Virus Scanners

Once a malware has been detected, it is possible to extract a signature from it (signatures are further discussed in chapter 3). Virus scanners detect malicious code using these signatures. The signatures are organized in databases, which the scan engine use to search predefined groups of files or system areas like the boot sector for malware. When there is a match for a given signature the entity in which it was found is reported as malware. At Ikarus the scan engine T3 is developed and maintained.

The major advantage of virus scanners is that they allow you to check programs before they are executed.

There are two basic kinds of scanners: on-demand and on-access. On-demand scanning is executed only at the user´s request. On-access scanners are memory resident. They load as a simple application and hook interrupts related to file and disk access, or they are implemented as device drivers that attach themselves to file systems. Typically files are scanned when they are opened, created or closed.

Early virus scanners used simple pattern matching techniques to identify malware. When malware writers began to challenge signature based virus scanners by polymorphism and other obfuscation techniques, they forced anti-virus vendors to develop more advanced techniques for analyzing disks and files. Today pattern matching is still the base of all virus scanners, but various techniques have been developed to identify malware faster and more efficient.

The signature databases are huge nowadays and the biggest challenge of a virus scanner is to use its limited time wisely. No more than a few seconds should be spent scanning a file. There are various techniques to speed up scanning. Hashing, as described by Knuth [22] is a common way to speed up searching. Hashing might be done on the first byte/bytes of the signature. By being selective in what starting bytes the signatures contain this can speed up scanning considerably. The T3 scan engine at Ikarus uses this technique. Signatures are saved in a hashtable with the first two bytes as key. When the scan engine parses a file, it checks every 2-byte combination of the file's machine code to see if it is present in the hashtable. Only when a 2-byte combination has been found in the hashtable will a full signature match be done. Autosig also takes advantage of this property to generate better signatures, as described in section 4.4.2.

Certain rules can be applied to keep the scanner from having to scan from one end of a file or system area to the other. The scanner should only scan segments likely to contain malicious code. If a scanner can locate the specific point in a file or system area where malicious code is situated, valuable time can be won. For example in structureless binary executables such as DOS or COM files, the scan engine can follow the various instructions that transfer control (such as JUMP and CALL instructions) and start scanning at the location to which such instructions point. The T3 scan engine at Ikarus only scans parts of the files. Which parts is chosen by running the files in a simulator to identify parts that might be infected or typical for malware. To this end the scan engine tries to identify instructions that transfer program control or program code in sections where it normally does not belong, such as data sections. These sections are called blocks.

The signature database at Ikarus contains a table of *known headers*. Some file types, such as PDF or RTF, are known to be somewhat "immune" to malware. To keep scan times down, the Ikarus scan engine avoids scanning such files. A PDF for example will always start with the bytes `255044462D`. If the scanner finds these bytes at address `0`, scanning is terminated. The same goes for the starting bytes `7B5C5C727466` that identifies a RTF file. At the time being 108 such known headers are kept in the database.

Window of vulnerability

Virus scanners can only find malware they are looking for. One major drawback with this is that the virus scanners need constant updates. Virus scanners are necessarily a step behind the latest crop of malware since there is a lot that has to happen before the scanner has an up-to-date signature database:

- Malware has to be detected somehow to begin with. Since virus scanners will not detect new malware, it will have some time to spread before someone detects it by other means.
- The newly-discovered malware must be analyzed and a suitable signature has to be extracted and tested.
- The signature must be incorporated into the next release of the signature database.
- The signature database must be distributed to the customers.

In the case of retail software, the software must be sent to be packaged, to the distributors, and then on to the retail outlets. Commercial retail software takes so long to get to the shelves, that it is almost certainly out of date. Virtually all product makers today provide some way to obtain updates via the Internet in order to help speed up the update process.

The window of vulnerability equals the time an anti-virus vendor needs to respond to new malware. That is, how long it takes to generate and distribute a signature for new malware. There are many metrics by which to measure the efficiency and effectiveness of an anti-virus product. This response time is one commonly used metric.

To compensate for this weakness most virus scanners are also capable of some heuristic analysis.

Heuristics

The term heuristic comes from the Greek word *heuriskein*, meaning ”to discover”, and describes the technique of approaching a problem through past experience, or knowledge.

When a file is scanned with a signature, it is determined to be either a specific malware or not. The result is easy to interpret and signature scanning is pretty accurate. Heuristic techniques, on the other hand, are working on the probability of a file being infected. A heuristic scanner looks for behaviors and indicators that malware use. This may include code to check a date, an oversized file, or attempts to access address books. Any of these indicators could simply be an innocent application, but when a number of such occurrences are combined, it may indicate malicious code. With heuristic scanners there is a thin line between being too stringent and producing a lot of false positives, or too lenient and missing malicious code.

Heuristics can also be used in a reverse manner, looking for behaviors that could not be viruses. In some cases, it is faster to determine that a file could not be malware, then finding one that could. For example, the W32/Simile virus, contains not only the payload, but code to compile, decompile, encrypt and decrypt the virus, resulting in a bigger file. The infection is 32K to 130K in length, so a file with size 25K could not contain that virus.

Heuristic search is supported by the virus scanner at Ikarus, but the results are to this date not presented to its users. In most cases the results are too unexact and users do not know how to respond to a message stating that ”this file *might* be infected with some unknown virus”. This problem is not unique for Ikarus.

2.4.2 Behavior Blockers

Unlike heuristics or virus scanners, behavior blocking software integrates with the operating system of a host computer and monitor program behavior in real-time for malicious actions. The behavior blocking software then blocks potentially malicious actions on the system. Monitored behaviors can include:

1. Attempts to open, view, delete and/or modify files.
2. Attempts to format disk drives and other unrecoverable disk operations.
3. Modifications to the logic of executable files, scripts or macros
4. Modification of critical system settings, such as start-up settings.
5. Scripting of email and instant messaging clients to send executables.
6. Initiation of network communications.

If the behavior blocker detects that a program is initiating some behavior that could be malicious, it can block these behaviors in real-time and/or terminate the responsible program. This gives it an advantage over virus scanners or heuristics. While there are literally trillions of different ways to obfuscate the instructions of a virus or worm, many of which will evade detection by scanners or heuristics, eventually malicious code must make a request to the operating system. Given that the behavior blocker can intercept all such requests, it can identify and block malicious actions regardless of how obfuscated the program logic is.

Watching software as it runs in real-time has some major drawbacks. Since the malicious code must actually run on the target machine before its behaviors can be identified, it can cause a great deal of harm before it has been detected and blocked. For instance, a virus might shuffle a number of seemingly unimportant files around the hard drive before infecting a file and being blocked. Even though the actual infection was blocked, the user may be unable to locate her files.

Behavior blockers can also have problems identifying for example trojans, since these do not have an easily characterized set of behaviors like viruses or worms. This in turn has lead to many false positives in the past, which is one of the reasons why behavior blocking only have had limited success on the market [26].

2.4.3 Intrusion Detection Systems

Intrusion Detection Systems (IDSs) sniff network traffic and inspect both the traffic flow and its content. They are an important part of network security. An IDS analyze network flow and protocol anomaly or use network signatures. Some IDSs combine both methods. They can be network- (NIDS) or host-based (HIDS).

The network flow and and protocol analyzer is practically a heuristic engine. It can for example have knowledge of the most relevant protocols, such as HTTP, FTP and SMTP, and match any anomalies in the protocols. The CodeRed worm, which sends its code as an HTTP request can be detected by a protocol analyzer due to its overly long URL [31].

The signature analyzer module looks for signatures in the network data. Signatures can be used to analyze protocol headers or to match a sequence of bytes in the data within the network packets.

IDSs have the drawback that they reduce performance. In order to correctly analyze traffic, all network packets have to be reassembled and this process can be very performance intensive.

2.5 Distribution of Malware

Malware propagates through a number of ways. In the early days, before Internet was as widespread as today, removable media such as floppy diskettes was the primary distribution channel. Today most malware is distributed electronically via email, either through an attachment or through code embedded in the message body. Infected applications or infected code on websites are other common distribution channels.

Other spreading mechanisms have also been observed. In June 2004 for example the SymbOS/Cabir worm, that run on Nokia 60 series phones running the Symbian operating system, appeared. It spread via the Bluetooth feature of wireless phones.

2.5.1 Malware Propagation

Classic malware uses a viral distribution pattern, in which one infected system infects another, and an epidemic develops. An outbreak of this type grows gradually and peaks after several days (see Figure 2.1). Viral distribution patterns allow anti-virus vendors valuable time to produce and distribute signature updates. The slower the propagation, the smaller the window of vulnerability, discussed in section 2.4, is.

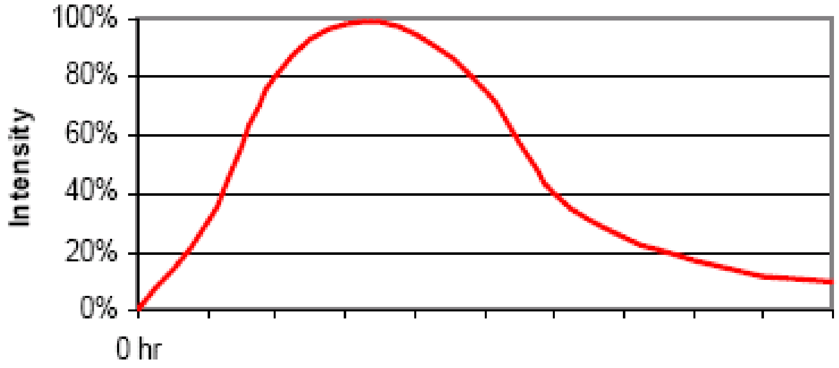


Figure 2.1: Typical viral propagation

However, new malware distribution patterns, such as short-span attacks and serial variant attacks have been seen. In these cases malware is distributed in ways that enable attacks to be executed fully before they can be blocked by signatures. Widespread adoption of these distribution methods could pose a serious threat to signature-based protection methods.

Typically, an entire short-span attack is completed within a few hours, sometimes within as little as 20 minutes. Unlike viral-propagation attacks, which die slowly, short-span attacks have a rapid buildup, steady distribution rate throughout the attack, and almost instant dropping off (see Figure 2.2). In many short-span attacks, anti-virus vendors avoid the trouble of developing a signature, since it will be obsolete by the time it is released.

A severe short-span attack was W32/Beagle.BQ, which started and finished within seven hours. Of 20 major anti-virus products tested independently by VirusTotal (a tool for scanning of suspicious files using several anti-virus engines), 10 did not manage to produce a signature before the end of the outbreak. 24 hours later, seven anti-virus vendors still had no signature for it at all.

Serial variant attacks uses the short-span characteristic, but extend it by a cumula-

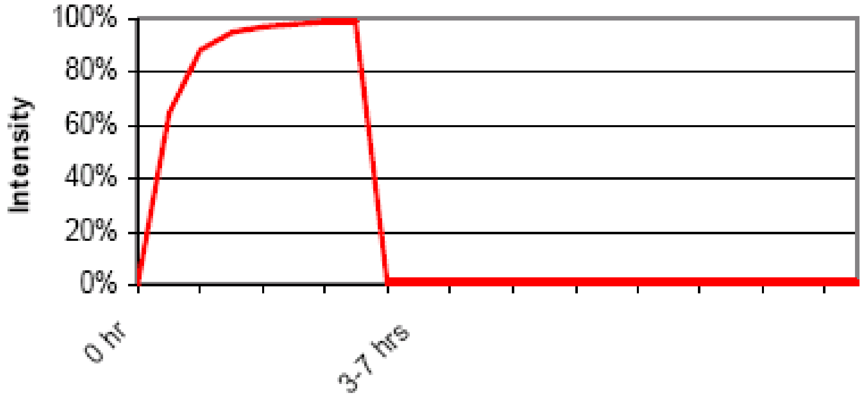


Figure 2.2: Short-span attack

tive factor. A series of variants of a certain malware, prepared in advance, are launched at closely-spaced time intervals. Each of the variants requires a new signature. The overall window of vulnerability of the attack is the cumulative vulnerable time span of the individual variants (see Figure 2.3). Theoretically, if an unlimited number of variants could be released, it would mean extending the window of vulnerability indefinitely.

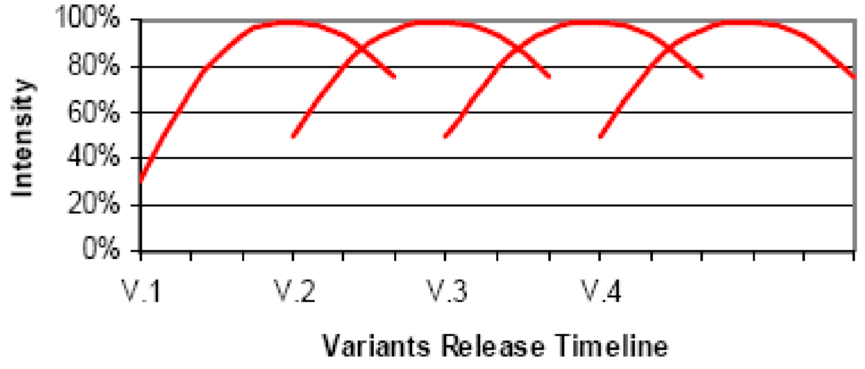


Figure 2.3: Serial variants attack

One example of a serial variant attack is W32/MyTob, which released, on average, one new variant every day over the course of six months in 2005. Even though the functionality of the different W32/MyTob variants was identical, a new signature had to be produced for each one. Another example is the W32/Beagle.BB-W32/Beagle.BF attack of 1 March 2005. During this attack 15 different new variants were released in a single day, almost one new variant per hour [14].

These new propagation patterns all challenge signature-based virus scanners, if new methods to faster generate signatures are not developed.

Chapter 3

Signatures

Suppose that we wish to determine whether a particular program P is infected with a virus V with the help of a virus scanner. The most naive method would be to scan the machine code of P, looking for bytes that exactly match V. Given the number of malware present, the amount of memory required just to contain all malware code would be huge. Not to mention the amount of time scanning would take. The solution to this problem is using a substring of V, a signature.

This chapter describes some basic properties of signatures and the workflow at Ikarus Software GmbH when generating them.

3.1 Basic Properties of Signatures

A signature is a sequence of bytes extracted from the body of a specific malware strain. It is generated by identifying a piece of bytecode, unique for the malware in question but unlikely to appear in other files.

Signatures have to possess two basic properties; they have to identify the malware from which they are generated and they should not identify any other files. That is, they should raise neither false positives nor false negatives. New signatures must be tested with the virus scanner in different environments before they are publicly released. They may otherwise trigger/reveal previously undiscovered bugs in the scanner.

False Positives A false positive exists when a test incorrectly reports that it has found a positive result where none really exists [36]. In anti-virus terms this means that malware has been found in a file which is not infected. An anti-virus software that keep finding malware in uninfected files will soon lose credibility with its users. This in turn might lead to that when malware really is found, the users do not care about the warning. Worse yet is when the anti-virus software starts trying to remove the supposedly infected file.

An example on what false positives can cause occurred on the 10th of March 2006. On this date McAfee released the file 4715.DAT, which contained a signature identifying a number of common commercial programs as the W95/CTX virus. The list of programs being reported as viruses is seven pages long and contains among others Microsoft Excel, Macromedia's Flashplayer, Adobe Update Manager and Norton's Ghost. These files were then quarantined, patched, and/or deleted, depending on the users local settings, leading to that the programs concerned stopped working.

False Negatives Exist when a test incorrectly reports that a result was not detected, when it was really present. Why this is bad hardly needs an explanation. An anti virus software unable to identify known malware obviously has a problem.

Bugs In April 2005 Trend Micro released a signature-file, which under certain conditions caused systems to experience high CPU power consumption [32]. To better identify the W32/Rbot worm family, Trend Micro had enhanced the decompression ability by supporting 3 new heuristic patterns. Unfortunately this enhancement caused performance issues when certain computer configurations were met. This incident shows the importance of testing signature-files before they are released.

3.1.1 Generic Signatures

Much malware start as a single infection, and through either mutation or modifications by other malware writers, it can grow into dozens of slightly different strains. In addition, some malware construction kits, such as the Virus Creation Laboratory or the Next Generation Virus Creation Kit, create similar viruses. There are also many databases with malware code available on the Internet [35, 34], where anyone can download malware code. By just changing a few lines of code a ”new” malware is created.

Rather than to generate a signature for every single strain, it can be advantageous to detect a malware family through a generic signature. In this case some invariant code among the set of variants has to be found. A byte sequence is then selected that is available in as many variants as possible to generate a signature from. These signatures often use wildcards where differences lie. Signatures generated from common code might possibly also identify future variants of the same malware family.

3.1.2 IDS Signatures

All of the most popular IDSs rely on signatures. These signatures can consist of byte sequences, just like the signatures used by virus scanners, or they can be a set of rules. When these rules are applied to an input stream, they will match every instance of the attack modeled by the signature. These signatures can for example restrain message length.

Systems that automatically generate signatures for Internet worms to be used by IDSs have been implemented. Autograph [20] relies on common characteristics of worms to classify network packages as benign or attack-bearing. COVERS [24] uses a forensic analysis of a victim server's memory to correlate attacks to inputs received over the network. Signatures that characterize inputs that carry attacks are then automatically generated. The system Polygraph [27] automatically generate signatures for polymorphic worms. Polygraph can generate multiple shorter byte-sequences as signatures. All the above take advantage of the fact that even in the case of polymorphic worms, some invariant code has to be present.

3.2 Signature Structure

Every anti-virus vendor generate signatures in their own form. In this section some general properties of signatures, and how these are implemented at Ikarus, are discussed. At Ikarus the signatures are organized in a database and have three important fields; name, type and bytepattern.

Name

A signature should have a name so that when a match has been found, the user can be notified which malware has been identified. The names should follow the CARO naming convention, discussed in section 2.3 as closely as possible. Ikarus use the same names as Kaspersky Labs [18].

Type

At Ikarus, signatures are classified by the type of infection they represent: boot sector, COM file, PE file, scripts or macros to mention a few. When a particular file is scanned, only the signatures that pertain to that file type is used. For example, a boot sector signature will not be used to scan a macro file. This helps keep scan times down.

Table 3.1 shows the different signature types used at Ikarus. SIG_ALL is an additional type that tells the scanner to search all possible files for the signature in question. SIG_ALL is allowed for testing purposes, but should not be used in real life, since it slows down scanning. Multiple types are allowed for files where more than one type might be fitting.

Bytepattern

The most essential part of a signature is the bytepattern as it is the actual code sequence identifying the malware in question.

Wildcards Some virus scanners allow wildcards in their signatures. Wildcards allow the scanner to detect if malicious code is padded with other junk code. Normally wildcards let the scanner skip bytes or ranges of bytes.

Mismatches Mismatches allow N number of bytes in the signature to be any value, regardless of their position. Mismatches are especially useful in generic signatures, but it is a rather slow scanning technique.

Table 3.1: Signature types at Ikarus Software GmbH

Signature Type	Target Files/Sections
SIG_MEM	System Memory
SIG_MEM_32	System Memory, 32 bit
SIG_BOOT	Boot sector
SIG_MBR	Master Boot Record
SIG_COM	DOS COM files
SIG_EXE_DOS	DOS Executables
SIG_EXE_NE	Win16 EXE or DLL
SIG_EXE_PE	Win32 EXE or DLL
SIG_EXE_LE	Win32 LE
SIG_MAC_WORD	Microsoft Word macros
SIG_MAC_EXCEL	Microsoft Excel macros
SIG_MAC_ACCESS	Microsoft Access macros
SIG_MAC_PPT	Microsoft Power Point macros
SIG_MAC_OTHER	Other macros in OLE format
SIG_JAVA	JAVA class files
SIG_BAT	DOS Batch files
SIG_JSCRIPT	JAVA scripts
SIG_VBSCRIPT	Visual Basic scripts
SIG_MIRC	mIRC Scripts
SIG_XSCRIPT	UNIX or Linux script
SIG_SCRIPT_OTHER	Other scripts, such as HLP
SIG_TEXT	Text files
SIG_ELF	Linux ELF files
SIG_TROJAN_DOS	Trojan Horses and Backdoors for DOS
SIG_TROJAN_WIN	Trojan Horses and Backdoors for for Windows
SIG_BIN	Other binary Formats

Some of the types might overlap and many are not used. More than 90 % of the signature are of type SIG_EXE_PE or SIG_TROJAN_WIN

At Ikarus, wildcards are allowed, identified with a > (jump X bytes forward from this position) or < (jump X bytes backward) followed by the number of bytes to be skipped. The } and { identifiers tell the scanner to skip an unknown number of bytes until the pattern after the identifier is found. No mismatches are allowed. The bytepattern part of the signature may also contain one or more checksums marked with a # identifier.

A signature for the trojan downloader W32/Banload.aap could for example have the following bytepattern:

```
7BC231D9898B3B2ED20CC0530803530C3BC27514D53126D8022F8B4367
412C00430C014604EB15033B59B7B685750DB7 >33 #0x78DD9178,144
```

This signature tells the scanner to match the 48 bytes starting with `7BC2` exactly. In this case `7BC2` will be searched for initially and only when these 2 bytes are found will a complete match be tried. The following 33 bytes are skipped and the 144 consecutive bytes adds up to the checksum `0x78DD9178`.

The following bytepattern is generated from the virus W32/Polip.a:

```
EC83EC18 }424438BE55DC3558BEC83EC04 }10151618BE55DC3
}47818BE55DC3558BEC83EC
```

In this case when the first four bytes have been found the scanner is told to jump forward until it finds the byte sequence starting with `4244` and ending with `EC04`. If this sequence is found it should jump forward again until the byte sequence starting with `1015` is found, and so on.

Signature Size

An important property, which has to be carefully chosen, is the size of a signature. The longer the byte sequence, the smaller the probability of false positives [25]. The shorter the signature, the more likely it is to be found in some perfectly legitimate code just by coincidence. However a very long signature will slow down scanning of files, since more bytes have to be matched. It also wastes memory.

One way to get around this, implemented at Ikarus, is using checksums. In this case a relatively small byte sequence is chosen and then a checksum over a bigger sequence is calculated and added to the signature. Only if the scanner finds the byte sequence in a file will it also calculate the checksum. This reduces the false positive probability, but keeps the signature database small. It does not reduce performance as the checksum is only calculated when the leading bytepattern has been found in a file.

The maximum total size of a signature at Ikarus is 500B, but most signatures are smaller than that.

3.3 Existing Workflow at Ikarus Software GmbH

To this date all signatures at Ikarus have been generated manually by the human analyst team. Lacking time to perform a deeper analysis, signatures are basically generated by trial and error. First a signature is generated based on routine and some standard methods developed through the years. The signature is then tested. Should a signature fail to pass a test, a new signature is generated and tested. This procedure is repeated until a working signature has been generated.

3.3.1 Incoming Malware

Ikarus Software GmbH has to deal with hundreds of new malware samples every day. The samples come from other anti-virus companies, customers, honeypots and many other sources. These are all stored in the Ikarus malware database, which today has reached the size of approximately 467 000 samples. For the purpose of sorting malware, Ikarus has developed a sorter tool that uses a number of scanners to classify and sort malware automatically. The sorter tool scans a (for Ikarus) unknown malware sample with several different scanners. If at least one of these already have a signature for this malware, Ikarus uses the malware name given by this scanner. Should different scanners assign different names, the most common one will be used.

The human analyst team search the malware database and make local copies of the malware for which they want to generate a signature.

One difficulty for the human analysts is new incoming variants of already existing malware, that is not found with already existing signatures. There is seldom enough time to compare all old and new variants with each other, in order to generate a generic signature. Instead new signatures are generated for new variants. This in turn can lead to the creation of more signatures than needed and signatures that sometimes overlap.

3.3.2 Signature Generation

After sorting and making local copies of the malware samples the scan engine is run over them to split the samples into block files. The block files represent the sections of the files the scanner deem likely to contain malicious code. It is from these block files that a signature then is generated. To possibly generate a generic signature the files are compared to find common code. The block files are opened with an internal tool that shows a hexadecimal representation of the machine code and highlights common code segments between the open files. Which block from a sample to compare with which block from another has to be chosen wisely. Normally the size of the original files together with the size of the generated block files can be helpful in finding which blocks might contain common code. For a malware where only a few samples exist or where all variants are nearly identical finding common code can be done pretty quickly. However, it is not hard see that when many samples exist and they have very little in common this can be tedious work to do manually. In the worst case scenario all possible pairs of files have to be examined just to find out that no similarities exist.

If common parts are found between the files, this code should be used for signature generation. A suitable byte sequence then has to be identified within the common code. As there is seldom enough time to analyze what the code actually does, the analysts use their experience to this end. First of all, actual program code is identified, since data sections are seldom suitable for signature generation. This is done by finding code that looks random to the human eye. It is then made sure that the first 2 bytes of the signature are as unique as possible. Starting bytes that should be avoided are for example `0x0000` or `0x9090`, since they appear very often in uninfected files. A byte sequence of about 48 bytes is then extracted. If more common code is present a CRC32 checksum calculated over this code and it is appended to the signature.

3.3.3 Tests

Each signature generated has to be tested before it can be distributed to the users of Ikarus anti-virus product. The newly generated signature is saved in a temporary signature database for this cause. First a false negatives test is performed by scanning the samples, from which the signature has been generated with this temporary signature database. If the signature finds the malware it is supposed to it has to be checked for false positives. Ikarus has a large collection of uninfected files for this purpose [1]. A scan of these files are initiated and, of course, no malware should be found using the current signatures. Since this test takes about half an hour, the signatures are not tested one by one. Instead this test is run when about 20 signatures have been generated.

At the time being about 10 percent of the manually generated signatures fail to pass the false positives test and have to be redone.

If a signature pass both these tests it is added to the signature database for distribution to end users. Updates are distributed according to need. Normally there will be at least one update a day, but if there is a malware outbreak more updates are normally needed.

[1] At the time being this collection has reached about 30 GB.

Chapter 4

Design of Autosig - An Automatic Signature Generator

The starting point for designing Autosig was to imitate the way human analysts generate signatures. During implementation some features were discovered that Autosig can use to improve its ability to generate signatures.

Autosig is primarily designed to be run over the collection of malware at Ikarus. When properly integrated with the other tools used by the human analyst team at Ikarus Software GmbH it can also be used in their daily work.

In this chapter the design principles of Autosig are explained.

4.1 Overall Design

It should be noted that Autosig is only automatic in the sense that the actual generation of signatures takes place without human intervention. The signatures are not automatically added to the updates that are distributed to users.

Autosig only generates generic signatures. For malware that exist in just one example in the Ikarus malware collection, no signatures are generated. Instead a CRC32 checksum is calculated over the whole file just as in the case of manual signature generation. If, at a later stage, other variants of this particular malware should be found, a new attempt to generate a signature will be made.

By taking advantage of the repeated false positives tests Autosig can be taught which code segments in a file not to use for signatures. By remembering these *negative signatures* they will be avoided when generating future signatures. It is anticipated that this will cause the initial false positive rate to drop.

Analyzing Malware

The most common approach when analyzing malware is reverse engineering. With this approach the function of the malware in question is analyzed and the malware specific code can be identified. This code, or parts of it, can then be used to generate a working signature. In the case of automatic signature generation reverse engineering is not sustainable. To automatically analyze a reverse engineered file a set of rules has to be implemented, stating what routines are typical for malware. Code segments including these routines are then characterized as malicious. This might work temporarily but it is highly unlikely that rules suitable for malware today will also be valid for the malware of tomorrow.

The scan engine at Ikarus only scans between certain addresses in a file, which further complicates this kind of analysis. A disassembler or decompiler can never return an exact copy of the program that after compilation produced the file being analyzed. This means that the addresses in the original file are different from those in the reversed engineered one. Since the scan engine does not scan whole files, generating a signature that perhaps is out of the scanned address span will cause false negatives. Thus, this kind of analysis should be left to the human experts.

Having said this, one might wonder how Autosig finds the code segments from which signatures are generated. Basically this is done by finding invariant code across two or more malware samples. Based on statistical data, the code segments most suitable for signatures are chosen from the invariant code. Then the same trial and error principle used by the human analysts is used.

4.2 Preparation Steps

Signatures generated by Autosig have to be compatible with the T3 scan engine used at Ikarus. The malware to be parsed by Autosig is copied from the malware database to a temporary directory. Before we can start parsing the malware samples the T3 scan engine has to generate block files from the samples in the temporary directory. These block files correspond to the addresses between which the scan engine will scan. While generating block files a log file is generated, stating which type the scan engine assigns the malware in question to. The temporary directory and all its files are to be deleted after execution.

4.3 Finding Invariant Code

The files to be compared are all in binary format. Many tools for finding binary *differences* between two files readily exists. Bsdiff [5], diffbin [13] and xdelta [38] are some examples of such programs. These are most commonly used for patching programs. The invariant code between two files can be found by inversing the result of one of these tools. To find the invariant code between an arbitrary number of files, all possible file-pairs have to be compared to another. The time such a comparison takes grows exponentially with the number of files being compared. Since there is no upper limit how many malware samples will be compared to another, it has turned out that for the purposes of Autosig, the best solution is a custom algorithm.

Since the malware samples being compared are all of the same family, we can assume that some similarities exists. When this assumption holds it is only necessary to compare two files with each other and remembering the common code between these. In the next step, this common code will be searched for in the rest of the files. Finding a byte sequence in a file is significantly faster than comparing two whole files. Only in the worst case scenario, when no files have anything in common with the others, every file has to be compared with every other file.

If no invariant code is found across the malware samples, no signature is to be generated.

Multiple Signatures

It can be the case that it is not possible to generate one signature that finds all of the malware samples in question. Some code suitable for signatures might be found in half of the samples and some other code in the other half. In this case two signatures are generated by Autosig.

4.3.1 Special Cases

When writing the routine for finding invariant code, some special cases had to be considered.

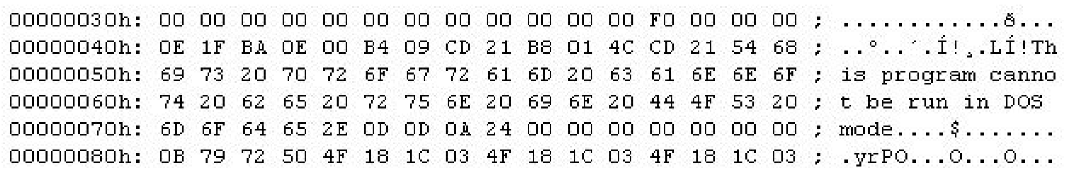
Exe packed files

An exe packer is a compression tool for executable files. Exe packers allow the compressed files to be executed without prior decompression. This is done by compressing an executable file and perpending a decompression stub, which is responsible for decompressing the executable and initiating execution. The decompression stub is a standalone executable, making packed and unpacked executables indistinguishable to users. No additional steps are required to start execution [36]. Some of the reasons for using exe packers are to reduce storage requirements, to decrease load times across networks and to decrease download times from the Internet. Exe packers are also often used by malware writers to prevent direct disassembly. Although it does not eliminate the chance of reverse engineering, exe packers can make the process more costly. Commonly used exe packers are ASPack, UPX and FSG.

Obviously, when comparing a compressed file with an original version of the file no common code will be found. This however is no problem that we have to worry about. When generating blocks from the original file, the T3 scan engine analyzes the file in a simulated environment. The files are unpacked, the interesting code segments are extracted and written to block files and these files are then later used to find common code.

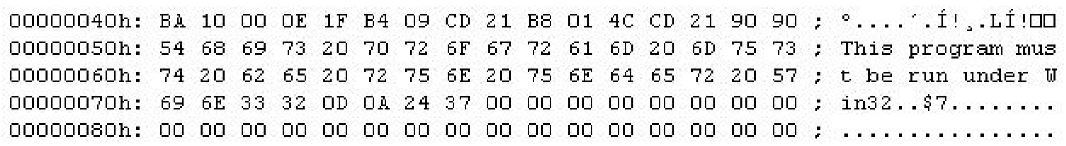
DOS stubs

At the beginning of every PE file there is a header containing DOS a stub [12]. This is just a minimal DOS EXE program that displays an error message (usually ”this program cannot be run in DOS mode” or “this program must be run under Win32” as shown in figure 4.1 and figure 4.2). The DOS stub remains as a compatibility issue with 16-bit Windows systems.



```
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00 ; ............ð...
00000040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ; ..º..´.Í!¸.LÍ!Th
00000050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F ; is program canno
00000060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 ; t be run in DOS 
00000070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 ; mode....$.......
00000080h: 0B 79 72 50 4F 18 1C 03 4F 18 1C 03 4F 18 1C 03 ; .yrPO...O...O...
```

Figure 4.1: DOS stub error message



```
00000040h: BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90 ; º....´.Í!¸.LÍ!□□
00000050h: 54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73 ; This program mus
00000060h: 74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57 ; t be run under W
00000070h: 69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00 ; in32..$7........
00000080h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
```

Figure 4.2: DOS stub error message

Because this header is present at the beginning of the file, some DOS viruses can infect PE images correctly at their DOS stub. Because of this a virus scanner has to scan this part of a PE file.

Consider the trojan horse Win32/VB.alc with original size 139 264B. One of its samples is divided in two blocks by the scan engine at Ikarus; one 112 byte big block consisting of the DOS header and one 32 768 byte block corresponding to some other code segment. In fact, when generating block files from any Win32/VB.alc sample, the result will always include this 112 byte block. When comparing these blocks to another the encouraging result that they are exactly the same will be found. However, using this as a signature would cause a flood of false positives, since it will be found in almost every PE file. Therefore it is not necessary to include these blocks in the search of common code. Before starting the comparison of files Autosig removes block files corresponding to DOS stubs.

Broken Samples

Incoming malware is sometimes corrupted on its way to Ikarus. These samples are not to be used for signature generation as they unlikely will contain meaningful code. Instead they will slow down the search of invariant code.

Broken samples normally differ a lot in their file size to the other samples of the same malware strain. Autosig uses this relatively naive approach and files that have a size much bigger or much smaller than the average of the files to be compared are removed before comparison.

4.4 Finding Code Suitable for Signature Generation

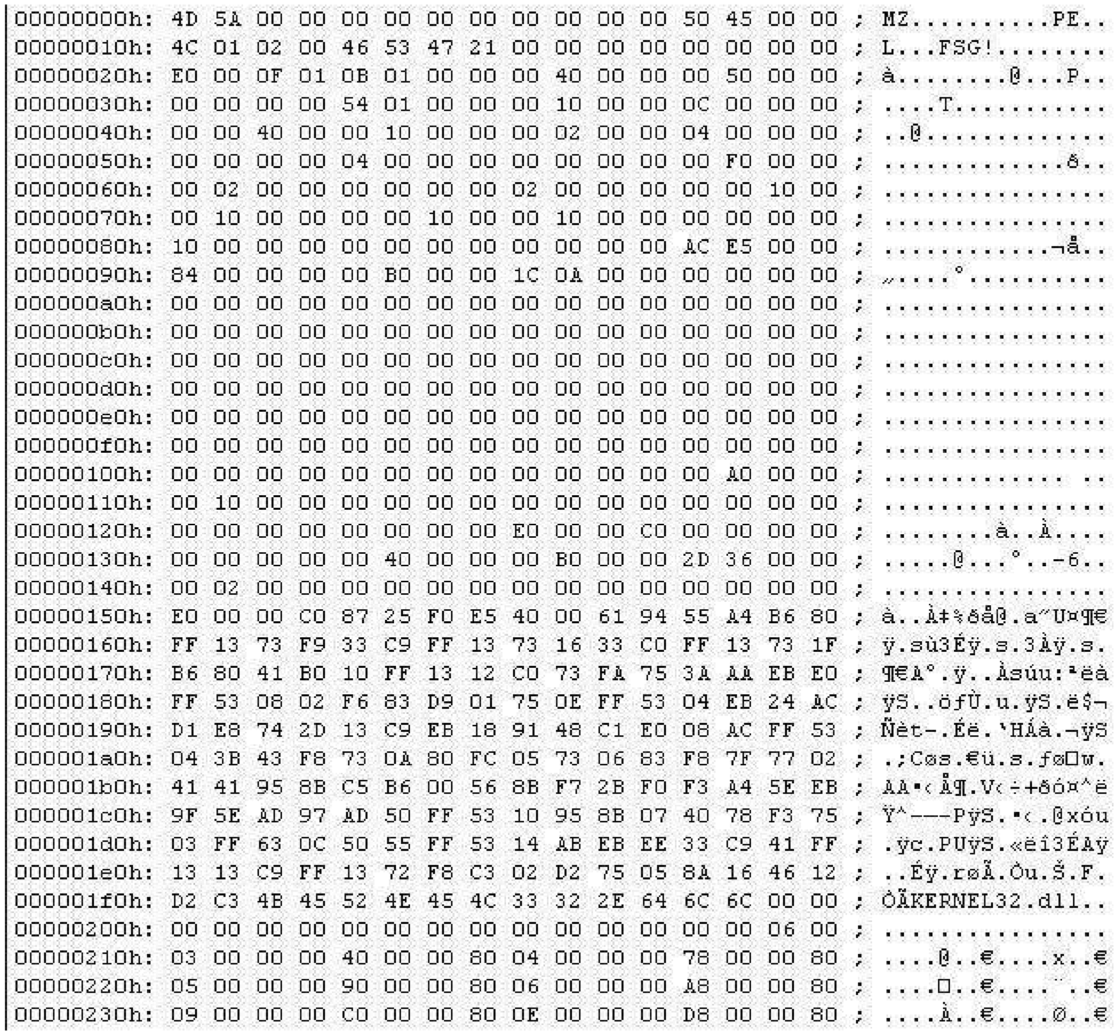
After finding invariant code between samples Autosig has to identify the parts of this code that are suitable for signature generation. In most cases big parts of the invariant code will also be present in uninfected files, leading to false positives if they are used for signatures. Exe packer code and data sections are such code segments and they have to be avoided when generating signatures.

4.4.1 Negative Signatures

Autosig performs a check against *negative signatures* to avoid code known to be not suitable for signature generation. This is for example old signatures that have caused false positives and code segments known to commonly exist in uninfected files. If the proposed signature code is equal to a negative signature it will be discarded. It will also be discarded if the proposed signature code is a substring of a negative signature. Figure 4.3 shows a code segment of a file packed with the popular exe-packer FSG. The bytes `0x46534721` at adress `0x14` states that this is a file packed with FSG.

The code from address `0x154` to `0x1f4` is the actual decompressor and this code will be found in every file packed with FSG. Since also legitimate files can be packed this way, generating a signature from this code, means deeming all files packed with FSG as malware. Therefore the decompressor code is saved as a negative signature, meaning that this code will never be used for signature generation.

The negative signatures are repeatedly updated with previously generated signatures, that have raised false positives. This way Autosig can be taught to avoid code which is not suitable for signatures and the false positives rate will drop successively, as shown in chapter 6.



```
00000000h: 4D 5A 00 00 00 00 00 00 00 00 00 00 50 45 00 00 ; MZ..........PE..
00000010h: 4C 01 02 00 46 53 47 21 00 00 00 00 00 00 00 00 ; L...FSG!........
00000020h: E0 00 0F 01 0B 01 00 00 00 40 00 00 00 50 00 00 ; à........@...P..
00000030h: 00 00 00 00 54 01 00 00 00 10 00 00 0C 00 00 00 ; ....T...........
00000040h: 00 00 40 00 00 10 00 00 00 02 00 00 04 00 00 00 ; ..@.............
00000050h: 00 00 00 00 04 00 00 00 00 00 00 00 F0 00 00 00 ; ............ð..
00000060h: 00 02 00 00 00 00 00 00 02 00 00 00 00 00 10 00 ; ................
00000070h: 00 10 00 00 00 00 10 00 00 10 00 00 00 00 00 00 ; ................
00000080h: 10 00 00 00 00 00 00 00 00 00 00 00 AC E5 00 00 ; ............¬å..
00000090h: 84 00 00 00 00 B0 00 00 1C 0A 00 00 00 00 00 00 ; „....°..........
000000a0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000b0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000c0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000d0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000e0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
000000f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000100h: 00 00 00 00 00 00 00 00 00 00 00 00 00 A0 00 00 ; ............. ..
00000110h: 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000120h: 00 00 00 00 00 00 00 00 E0 00 00 C0 00 00 00 00 ; ........à..À....
00000130h: 00 00 00 00 00 40 00 00 00 B0 00 00 2D 36 00 00 ; .....@...°..-6..
00000140h: 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000150h: E0 00 00 C0 87 25 F0 E5 40 00 61 94 55 A4 B6 80 ; à..À‡%ðå@.a”U¤¶€
00000160h: FF 13 73 F9 33 C9 FF 13 73 16 33 C0 FF 13 73 1F ; ÿ.sù3Éÿ.s.3Àÿ.s.
00000170h: B6 80 41 B0 10 FF 13 12 C0 73 FA 75 3A AA EB E0 ; ¶€A°.ÿ..Àsúu:ªëà
00000180h: FF 53 08 02 F6 83 D9 01 75 0E FF 53 04 EB 24 AC ; ÿS..öƒÙ.u.ÿS.ë$¬
00000190h: D1 E8 74 2D 13 C9 EB 18 91 48 C1 E0 08 AC FF 53 ; Ñèt-.Éë.‘HÁà.¬ÿS
000001a0h: 04 3B 43 F8 73 0A 80 FC 05 73 06 83 F8 7F 77 02 ; .;Cøs.€ü.s.ƒø□w.
000001b0h: 41 41 95 8B C5 B6 00 56 8B F7 2B F0 F3 A4 5E EB ; AA•‹Å¶.V‹÷+ðó¤^ë
000001c0h: 9F 5E AD 97 AD 50 FF 53 10 95 8B 07 40 78 F3 75 ; Ÿ^—PÿS.•‹.@xóu
000001d0h: 03 FF 63 0C 50 55 FF 53 14 AB EB EE 33 C9 41 FF ; .ÿc.PUÿS.«ëî3ÉAÿ
000001e0h: 13 13 C9 FF 13 72 F8 C3 02 D2 75 05 8A 16 46 12 ; ..Éÿ.røÃ.Òu.Š.F.
000001f0h: D2 C3 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 ; ÒÃKERNEL32.dll..
00000200h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 00 ; ................
00000210h: 03 00 00 00 40 00 00 80 04 00 00 00 78 00 00 80 ; ....@..€....x..€
00000220h: 05 00 00 00 90 00 00 80 06 00 00 00 A8 00 00 80 ; ....□..€....¨..€
00000230h: 09 00 00 00 C0 00 00 80 0E 00 00 00 D8 00 00 80 ; ....À..€....Ø..€
```

Figure 4.3: FSG decompressor code

4.4.2 Bad Starting Bytes

To speed up scanning of files, the scan engine at Ikarus uses hashing on the first two bytes of a signature. The scanner checks the first two bytes of a signature before checking the whole signature. Only if these bytes match, the following bytes are checked for a full match. The comparison ends with the first mismatch. By using statistics on how often any 2-byte combination occur in legitimate files, Autosig avoids signatures starting with bytes that occur often in these. This is not only a way to optimize the signatures generated but also to speed up scan times. A signature starting with a 2-byte combination that occurs very often in uninfected files will cause a full signature match as often. Matching a whole signature is obviously more time consuming than just checking two bytes in a hashtable. Table 4.1 shows some 2-byte combinations in hexadecimal representation and their corresponding values in the statistics used by Autosig.

Table 4.1: 2-byte combination prevalence in uninfected files.

2-byte combination	Value
0100	113
4000	290
6A00	192
9F00	3
E800	2044
FC00	10

These statistics are produced by scanning a big collection of uninfected files with the current signature database. It should be noted that the resulting statistics are correlated to the signature database. The values shown in table 4.1 is not the actual value of how often a particular 2-byte combination occur, but a weighted value. As an example the bytes `0x000` appears extremely often in every file, but is not represented in theses statistics because no signature starting with this value is allowed.

When designing Autosig the choice had to be made weather these 2-byte combinations should be hard coded into the program or loaded dynamically at run-time. Since new operating systems or programs could alter these statistics the choice fell on dynamical loading. Whenever new files are added to the collection of uninfected files the user can choose to renew the statistics. Some values not present in these statistics, such as `0x0000`, have to be hard coded in to the program.

Chapter 5

Implementation of Autosig

Autosig is a tool for automatic generation of malware signatures. It is a command line tool that runs on Windows, written in C++.

Autosig is a single-threaded program performing its tasks sequentially. It parses directories recursively, meaning that the directory given as input can in its turn contain directories and so on. Program execution is schematically shown in figure 5.1.

Even though time is a crucial component when generating signatures, since Autosig is not supposed to be run in real-time, there are no execution time requirements to be met. It does not really matter if the generation of a signature takes 2 seconds or 2 minutes. However an upper limit of 8 minutes has been set. This limit should never be reached, but since there is no limit in how many files that can be given as input, the calculation time has to be controlled in some way.

When comparing files memory usage is often an issue, since it is normally needed to keep the whole files in memory. Autosig will be run on the computers at Ikarus Software GmbH where enough memory for these purposes is available. Therefore saving of memory has not been a primary goal when implementing Autosig.

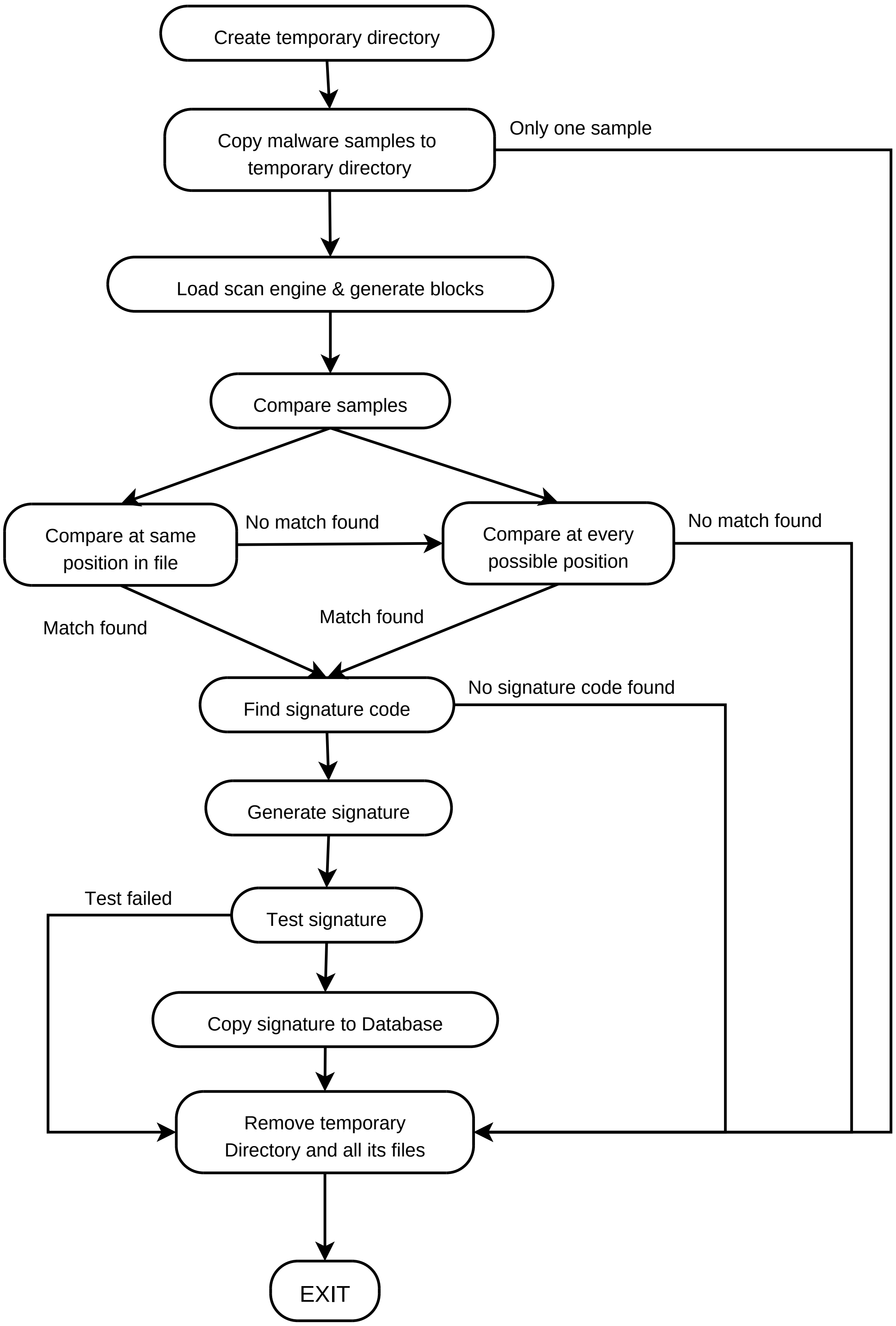

Create temporary directory
Copy malware samples to temporary directory
Only one sample
Load scan engine & generate blocks
Compare samples
Compare at same position in file
No match found
Compare at every possible position
No match found
Match found
Match found
Find signature code
No signature code found
Generate signature
Test failed
Test signature
Copy signature to Database
Remove temporary Directory and all its files
EXIT

Figure 5.1: Program flow

5.1 External Dependencies

Autosig generates signatures for the T3 scan engine at Ikarus. Because of this some features implemented by the scanner have to be adopted by Autosig. This is a limitation, but also a benefit. A limitation because the samples to analyze have to be split up in blocks before they can be parsed, complicating the analysis of them. A benefit because Autosig can use some of the functionality of the scan engine. Most of all being able to test and evaluate the signatures are very beneficial. Autosig uses the scan engine as a shared library, a Windows DLL. During execution the following functions are called:

```
T3Init              //initialize scan engine
T3ScanLog           //split files in blocks
T3CreateVdb         //create a temporary signature database
T3VdbAddSignature   //add a signature to this database
T3Scan              //scan files
T3Exit              //exit
```

5.2 Find Invariant Code

As explained in the previous chapter, Autosig uses a custom algorithm to compare files for similarities. To start with a minimum length for common code segments is defined, which at the same time is the minimum length of a signature. For this purpose a global variable called `MIN_SIG_SIZE` is introduced. By default this is set to 32 byte. A 32 byte value is long enough to possibly be unique in a 32-bit file and small enough to not waste memory. Therefore it could theoretically be used as a signature.

5.2.1 Algorithm

Autosig uses a combination of hashtables and a slightly modified version of the well known Knuth Morrison Pratt (KMP) string search algorithm to find invariant code between files.

The simplest and least efficient way to find a pattern P within a text S is to check each byte, one by one. This approach has a worst case performance of $O(n * m)$, where n is the length of S and m is the length of P. Fortunately, many fast string searching algorithms exist, such as the KMP [21] or Boyer-Moore algorithm [4]. The KMP algorithm searches for occurrences of P within S by employing the simple observation that when a mismatch occurs, the pattern itself embodies sufficient information to determine where the next match could begin. Thereby re-examination of previously matched characters is avoided. The algorithm has complexity $O(n)$. Boyer-Moore is a slightly faster algorithm with an average complexity of $O(n/m)$ and only in worst case $O(n)$. Normally string searching algorithms return exact matches. However, for the purposes of Autosig it is enough if a match bigger or equal to `MIN_SIG_SIZE` is found. The Boyer-Moore algorithm starts searching with the last character of P and skips the preceding characters when a mismatch occurs, which is precisely what makes the algorithm so fast. Due to this fact it is not suitable for the purposes of Autosig. Those skipped bytes could be a smaller match, perfectly suitable for signature generation. Instead the choice of algorithm fell on KMP, which is also relatively easy to implement.

If the given path only contains one malware sample, Autosig exits. Until all files have been parsed the following steps are executed:

Step One

The first step is to compare two whole files (in the following, a ”file” means all block files corresponding to the original file). The biggest file, in the following called file 1, is set to be the base file to compare the other with. The second biggest file, file 2, is chosen to start comparing with. If file 1 and file 2 have exactly the same size it has turned out that they often have common code at the same addresses. Knowing this we can perform a first byte-to-byte check, which is a lot faster than looking for similarities at every possible position. If matches are found we step directly to step four.

Step Two

If step one failed a checksum for every 32 byte value in file 1 is calculated. These checksums are used as keys in a hashtable where the address in file 1 of the first of the 32 byte is the value. Since it is not unlikely that a 32-byte value will repeat itself in a file, duplicate keys are allowed. Then file 2 is opened and parsed byte by byte, calculating the same checksum. The checksum is used to retrieve the value in the hashtable, i.e. the position of this checksum in file 1. The positions in both files are saved as an Integer Pair in a vector.

Step Three

When file 2 has been parsed, the vector containing the position pairs is evaluated. As long as the difference between both the first and the second value of a pair and its precursor is 1, we have found common code.

If no matches are found step one is repeated with the next biggest file, which will replace file 2. Should it be the case that file 1 has nothing in common with any other files, it is simply ignored and the procedure is repeated with file 2 replacing file 1.

Step Four

When reaching this step common code has been found and we create a new instance of a code struct, as shown below.

```
struct code {
    char *file;
    int start, size;
    vector<char *> files;
    bool operator< (const code a)
    {
        return this->files.size() < a.files.size();
    }
};
```

This struct contains information on which file has been used as file 1, the starting position in this file, the size of the common code found and in which other files the code is present. Only code segments bigger than `MIN_SIG_SIZE` are considered. This struct is added to a list.

Step Five

Now we can begin looking for the common code sections in other files. This is done by opening the next biggest file, file 3, and reading it in to a char array. Then the list of common code is parsed, node by node. The code segment represented by the code struct is read in to another char array. Using a slightly modified version of the KMP search algorithm it is checked if the code segment is present in file 3. If an exact match is found, file 3 is added to the file list in the struct. If a smaller match is found the size is also set to the new, smaller value. This step is repeated until there are no more files to compare with.

If there are files that have nothing in common with file 1, step one is repeated with only these files, the biggest of them as file 1.

5.2.2 Sorting List of Common Code

Upon successful execution of the above algorithm all common code segments have been put in a list. This list will later be used for signature generation but first it has to be sorted, which is done by ranking the code segments. The common code found in most different files is ranked highest, since a signature generated from this code would find the most malware. For example, when comparing five malware samples it could be that some code segment is found in all of the samples, some in four of the samples and some other code segment is found only in two. In this case the code segment found in all samples will be ranked the highest and the one found in two the lowest.

Of course there might exist code segments that are found in equally many files. How these are ranked internally is decided by which flag the user has entered upon starting Autosig.

Sort After Code Segment Size (-ss flag)

If the user has entered the "-ss" flag the code segments with the biggest size will be ranked highest. This might help finding a good signature faster as the probability to find a good signature in a big code segment is higher than finding it in a small one. Picture 5.2 shows part of a list sorted with this option.

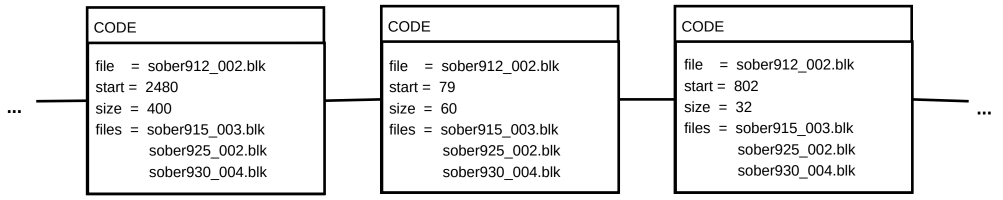


Figure 5.2: List sorted according to size

Sort After Code Segment Position in File (-ps flag)

When entering this flag code segments are put in a sequential order depending on their position in the file. Choosing this option should make scanning of files slightly faster, since the scan engine can parse the file from beginning to end without having to jump back and forth. Picture 5.3 shows the same list as above, sorted with this option.

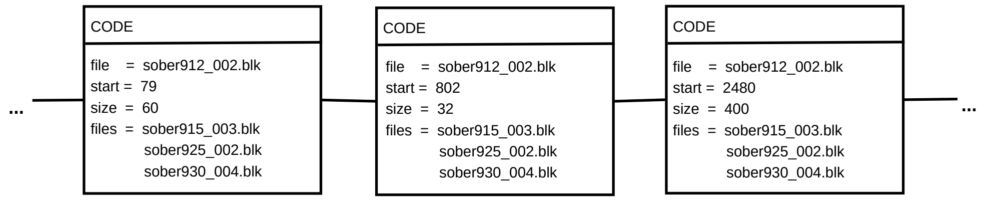


Figure 5.3: List sorted according to position

5.3 Find Good Signature Code

After setting up and sorting the list of invariant code, Autosig parses this list to sort out the code segments that are suitable for signature generation. To find good signature code bad starting bytes are eliminated and code segments known to be non-malware code are avoided.

5.3.1 Bad Starting Bytes

Statistics on how often 2 byte combinations occur in uninfected files are saved in a file that Autosig reads in run-time. There exists 65 536 different such combinations. The statistics given by the T3 scan engine consists of 35 862 different values, meaning that there are 29 674 combinations that never appear. These are the ones that should be used as starting bytes for a signature. It is not a guarantee that the signature will raise no false positives, but it increases the probability.

Following Bytes

It has turned out that these statistics are not only helpful in choosing good starting bytes. They can also be used to check the following bytes. For example to remove code segments consisting of just zeros. Autosig uses only code segments consisting of 2 byte combinations that have 8 hits or less in the statistics.

5.3.2 Negative Signatures

When Autosig has parsed the list of common code and found segments suitable for signature generation, this code is compared against the negative signatures. If the proposed signature code is found in this file it will be discarded.

5.4 Generate Signature

Upon successful completion, the result of the previous steps is a list with code segments suitable for signature generation. A signature is then generated as follows:

Signature number Every signature has to have a number. This is normally set automatically when adding the signature to the database, but a temporary number has to be set for the test database. This variable is saved in an external file and the number is incremented by 1 per signature.

Signature type The signatures also have to have a type compatible with the type given by the scan engine, as explained in section 4.2. The type is retrieved by reading the log file generated by the scan engine in the preparation steps. It states which malware type the scanner assigns the malware in question to. The log file often contain multiple types, but these are avoided by Autosig when possible. It is assumed that if the type `SIG_EXE_PE` or `SIG_TROJAN_DOS` is present, this will be adequate. Otherwise multiple types is used.

Signature name Autosig uses the name of the malware given as input as signature name. Since more than one signature might be necessary a number is appended to the name.

Bytepattern In this field the actual code segment found in the previous steps is situated. If the list from the previous steps contain more than 1 node, the first one is used as a pure bytepattern and the rest of the nodes are used to calculate up to 20 checksums. More than 20 checksums is normally not necessary and would only waste memory. Should the first node be bigger than `MIN_SIG_SIZE` a CRC checksum is calculated over the exceeding code segment.

5.5 Tests

Before any signatures are ready to be shipped to the users of Ikarus anti-virus software it has to be thoroughly tested. The finished signatures are written to a file and from this a temporary signature database is generated. The directory with the malware, from which the signatures have been generated is then scanned with the temporary signature database. All malware from which a signature has been generated should be found during this scan. Chapter 6 will show that the signatures generated by Autosig hold in 99 per cent of the cases. If the signature does not find the malware it is supposed to, it will be discarded.

At this point Autosig has finished its execution and exits after removing block- and log-files. The signatures also have to be tested for false positives, but this test must be initiated by the user. Signatures that do not pass this test will be (manually) added to the file with negative signatures as described in section 5.3.2.

Chapter 6

Test Results

Autosig has been thoroughly tested to identify and measure the benefits of automatic signature generation. Two different areas of use are of interest; signature generation from new malware samples on a daily basis and signature generation from old samples in the malware database. The latter is to possibly shrink the number of signatures in the database. Both these areas of interest have been tested. During the tests the following was observed:

False Negatives Rate Only one false negative was produced during all of the tests. This was due to a faulty log file from the scan engine, resulting in a signature type that was not correct. If the signature type is not correct the scanner will not search for the signature in the area were it can be found. After manually changing the signature type, the signature worked.

False Positives Rate A finished signature is not allowed to generate any false positives. However this is not a hard limit for Autosig since signatures causing false positives simply will be discarded as negative signatures, and then a new try will be done. This way the program should learn which code segments to avoid. As anticipated the false positives rate dropped after a few days, as more negative signatures are added, as seen in table 6.1.

Sort Algorithm When sorting the list of common code as described in section 5.2.2 two different options have been given to the user; sort according to size or position. However, it turned out that these options did not have much effect on the overall performance. No improvement of the signatures could be noticed in either of the cases, nor did any of the different algorithms yield any time benefits.

6.1 Daily Tests

Autosig has been tested on a daily basis on new malware samples. The tests have been performed (with some exceptions) between the 1st and the 24th of April 2006. During this period only new incoming malware was parsed. No consideration was taken to old samples of the same type of malware. The malware samples were copied to a directory with subdirectories so that all malware from one day could be parsed at the same time. Each malware family (and every possible variant) were put in own subdirectories. The results from these tests can be found in table 6.1.

As can be seen in the table these tests were not performed every day, due to time constraints. The total number of samples stated does not include those malware families where only one example was present. The column "Signatures" states how many signatures were generated and "Samples Found" states how many samples were found with these. "Runs" means how many times Autosig had to be run over the samples before no false positives were generated. That is, how fast did Autosig learn which code to use for signatures.

Table 6.1: Daily Tests April 2006

Date	Samples	Signatures	Samples Found	False Neg.	Runs	Time(s.)
01	64	11	29	0	5	236
02	18	2	6	0	1	45
04	182	18	41	0	3	235
05	96	16	38	0	2	126
06	108	19	48	0	1	137
07	187	26	137	0	2	389
08	49	9	27	0	4	66
09	44	6	15	0	1	32
10	268	22	198	0	2	198
11	167	13	80	0	1	214
12	62	15	32	0	1	182
13	30	4	14	1	2	195
14	189	18	69	0	1	639
15	36	6	14	0	1	88
16	42	5	14	0	3	53
17	141	11	75	0	1	158
22	117	15	73	0	1	144
23	119	8	89	0	1	472
24	328	7	305	0	1	149

The initial false positives rate sank pretty fast after the first week of tests, to a near minimum at the end of the month. This shows that Autosig was able to learn which code segments to avoid. The first day Autosig had to be run as many times as 5 to be able to sort out the code to avoid. This was because a substring was extracted from a relatively big code segment also present in morpheus.exe, a file sharing client. Since a signature was only generated from a part of this code, when adding the signature to the negative signatures there were still morpheus-code left that Autosig tried to generate a signature from, leading to yet another false positive.

Another encouraging result of these daily tests was that some of the signatures generated managed to identify new samples from the same malware family. For example the signature

```
B12F6E70DA2746823D12BA24F4CAFF3B33CE4BD20F7F4419FB058B09997
DD0DBC5CF2057AA3E0F78A95322CA68787506 >10 #0x19d1e599,108
>1 #0xa90b914b,740 >4 #0x507a8328,108
```

generated from the Backdoor trojan W32/IRCBot.pu on the 2nd of April also found the new W32/IRCBot.pu samples arriving on the 3rd, 12th, 13th and 14th. This shows that the signatures generated by Autosig are indeed able to find new examples of already known malware.

6.2 Database Tests

Ikarus has access to a malware collection of about 467 000 malware samples and during the years the human analyst team has produced circa 123 000 signatures for these. New variants of old malware are often added to the malware collection. In the signature database there are many overlapping signatures. A tool like Autosig can be used to minimize the overlapping signatures and thereby make it possible to remove signatures that are not necessary anymore. There are also some malware samples for which no signature exists.

For this purpose Autosig has been run over parts of the malware collection. With the help of a script every malware family where more than one sample is present in the malware database was parsed by Autosig. Only the main families were parsed together. For example, all W32/Sober variants (W32/Sober.a, W32/Sober.b and so on) were parsed together. Parsing subfamilies separately might give a different result.

With five 2GHz Xenon CPUs it took 2 days for the circa 350 000 samples. From these 2906 signatures were generated. 908 of these caused false positives. 739 signatures found other malware apart from that what they should find. Since this might mean that the signatures are not specific enough these were not added to the signature database. For 373 of the automatically generated signatures, there were already old signatures present that were deemed better.

At the end 886 new signatures were added to the signature database. Doing this, 12 000 old signatures could be removed. 10 000 malware samples, for which there was no signature present earlier, were able to be detected. Of the accepted signatures 99% were of the `SIG_EXE_PE` type. A large number of the `SIG_SCRIPT` signatures generated false positives.

During the test it was noticed that when more than 10 samples were parsed the time limit of 8 minutes was often reached.

6.3 Case Study: W32/Sobig

The first variant of the email worm Sobig appeared in January 2003. After that numerous variants have appeared and in Juli 2003 almost 50 % of all malware in the wild was from the W32/Sobig family.

Ikarus has 545 samples of the different W32/Sobig variants in its malware collection. When running Autosig over all variants one signature is generated with the following bytepattern:

```
4C98FFFF3BFE7410FF75F48B3D40404100FFD7FF75F0FFD73975087409FF
7508FF150C4041006A01 >5 #0xDD5B7F86,44 >42 #0x891F1CD6,164
>6 #0xF1793237,48
```

Parsing the 545 W32/Sobig samples was surprisingly fast and took only 37 seconds. This is probably because invariant code can be found at the same position in almost all files. A log file from this execution of Autosig can be found in Appendix A.

With this one signature 508 samples are found and it generates no false positives.

This result clearly shows one of the benefits of automatically generating signatures. Manually comparing all the block files generated from the 545 samples is a tedious work, better done by a computer.

6.4 Case Study: W32/Polip

W32/Polip is a polymorphic virus for the Windows platform that appeared in March 2006. It spreads through peer-to-peer networks and apart from a complicated polymorphic technique, the virus also has a function for turning off certain anti-virus and security programs.

Windows executables are infected by the virus, which writes the code of the polymorphic decoder into unused spaces of code sections. When doing this, the virus creates a new section and places its main encoded code there, moving the resource section, if any exists, below. When infecting a file it does not modify the original entry point, but replaces addresses of some randomly selected API calls with the start address of the virus.

The virus is memory resident and intercepts the following API functions - ExitProcess, CreateProcess, CreateFileA, LoadLibraryExA, SearchPathA, CreateProcessW, CreateFileW, LoadLibraryExW, SearchPathW. When any of these functions is called, new files get infected.

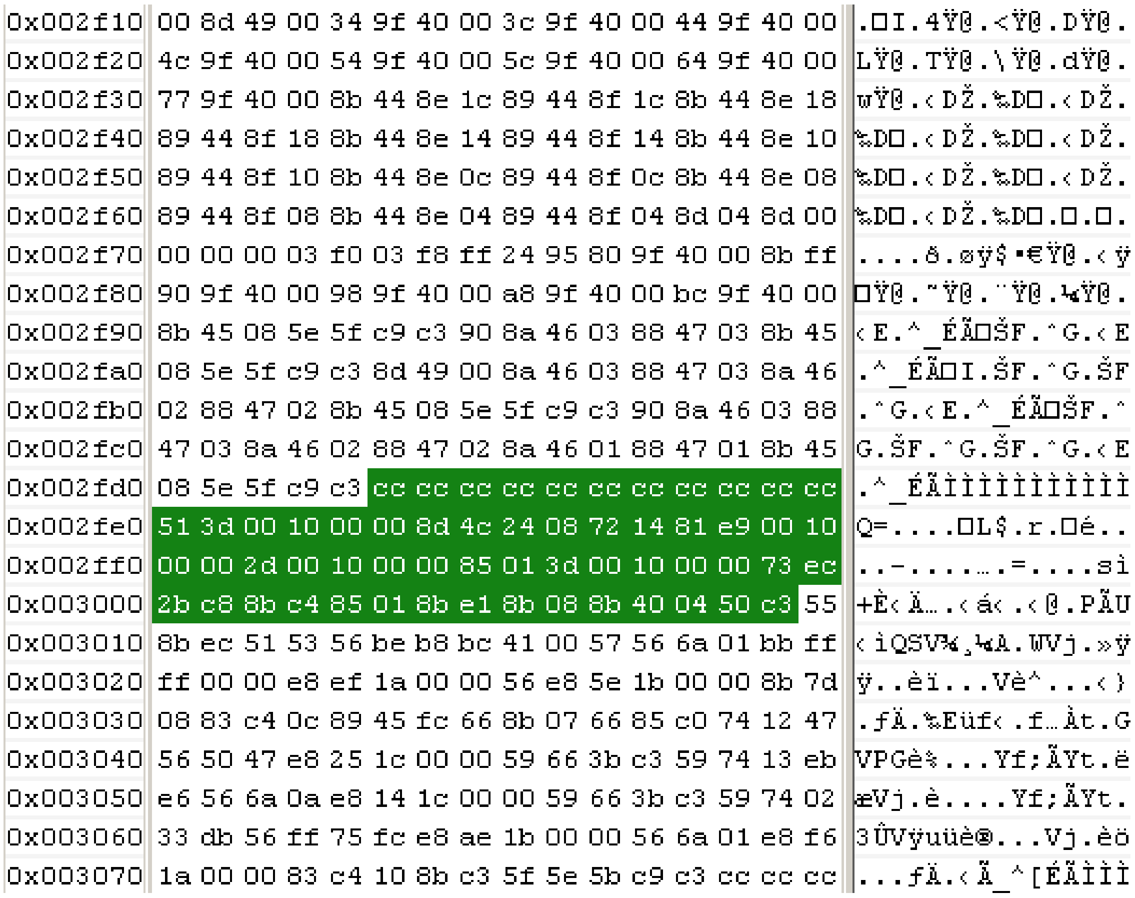
It took days for most anti-virus products, including Ikarus, to generate a working signature for this virus.

Autosig did not manage to generate any signature for W32/Polip, as anticipated in the case of polymorphic viruses. It did find some invariant code among two of the samples, shown in figure 6.1. The invariant code is highlighted in the picture.

This code was characterized as not suitable for signature generation. The leading `0xCCCC` bytes are obviously not suitable to start a signature with, as these are padding bytes that appear often in all kinds of files. To get a minimum sized signature of 32 bytes it has to start at address 0x2fef. However, none of the bytes `513D001000008D4C2408721481E900` preceding this address are suitable to start a signature with, as they all appear to often in the statistics.

A log file from this execution of Autosig can be found in Appendix B.

Figure 6.1: Invariant W32/Polip code



```
0x002f10  00 8d 49 00 34 9f 40 00 3c 9f 40 00 44 9f 40 00  .□I.4Ÿ@.<Ÿ@.DŸ@.
0x002f20  4c 9f 40 00 54 9f 40 00 5c 9f 40 00 64 9f 40 00  LŸ@.TŸ@.\Ÿ@.dŸ@.
0x002f30  77 9f 40 00 8b 44 8e 1c 89 44 8f 1c 8b 44 8e 18  wŸ@.‹DŽ.‰D□.‹DŽ.
0x002f40  89 44 8f 18 8b 44 8e 14 89 44 8f 14 8b 44 8e 10  ‰D□.‹DŽ.‰D□.‹DŽ.
0x002f50  89 44 8f 10 8b 44 8e 0c 89 44 8f 0c 8b 44 8e 08  ‰D□.‹DŽ.‰D□.‹DŽ.
0x002f60  89 44 8f 08 8b 44 8e 04 89 44 8f 04 8d 04 8d 00  ‰D□.‹DŽ.‰D□.□.□.
0x002f70  00 00 00 03 f0 03 f8 ff 24 95 80 9f 40 00 8b ff  ....ð.øÿ$•€Ÿ@.‹ÿ
0x002f80  90 9f 40 00 98 9f 40 00 a8 9f 40 00 bc 9f 40 00  □Ÿ@.˜Ÿ@.¨Ÿ@.¼Ÿ@.
0x002f90  8b 45 08 5e 5f c9 c3 90 8a 46 03 88 47 03 8b 45  ‹E.^_ÉÃ□ŠF.ˆG.‹E
0x002fa0  08 5e 5f c9 c3 8d 49 00 8a 46 03 88 47 03 8a 46  .^_ÉÃ□I.ŠF.ˆG.ŠF
0x002fb0  02 88 47 02 8b 45 08 5e 5f c9 c3 90 8a 46 03 88  .ˆG.‹E.^_ÉÃ□ŠF.ˆ
0x002fc0  47 03 8a 46 02 88 47 02 8a 46 01 88 47 01 8b 45  G.ŠF.ˆG.ŠF.ˆG.‹E
0x002fd0  08 5e 5f c9 c3 cc cc cc cc cc cc cc cc cc cc cc  .^_ÉÃÌÌÌÌÌÌÌÌÌÌÌ
0x002fe0  51 3d 00 10 00 00 8d 4c 24 08 72 14 81 e9 00 10  Q=....□L$.r.□é..
0x002ff0  00 00 2d 00 10 00 00 85 01 3d 00 10 00 00 73 ec  ..-.....….=....sì
0x003000  2b c8 8b c4 85 01 8b e1 8b 08 8b 40 04 50 c3 55  +È‹Ä….‹á‹.‹@.PÃU
0x003010  8b ec 51 53 56 be b8 bc 41 00 57 56 6a 01 bb ff  ‹ìQSV¾¸¼A.WVj.»ÿ
0x003020  ff 00 00 e8 ef 1a 00 00 56 e8 5e 1b 00 00 8b 7d  ÿ..èï...Vè^...‹}
0x003030  08 83 c4 0c 89 45 fc 66 8b 07 66 85 c0 74 12 47  .ƒÄ.‰Eüf‹.f…Àt.G
0x003040  56 50 47 e8 25 1c 00 00 59 66 3b c3 59 74 13 eb  VPGè%...Yf;ÃYt.ë
0x003050  e6 56 6a 0a e8 14 1c 00 00 59 66 3b c3 59 74 02  æVj.è....Yf;ÃYt.
0x003060  33 db 56 ff 75 fc e8 ae 1b 00 00 56 6a 01 e8 f6  3ÛVÿuüè®...Vj.èö
0x003070  1a 00 00 83 c4 10 8b c3 5f 5e 5b c9 c3 cc cc cc  ...ƒÄ.‹Ã_^[ÉÃÌÌÌ
```

6.5 Evaluation of Test Results

In this chapter some test results have been presented that clearly show the benefits of automatic signature generation. When more than one sample is available from a malware family and some invariant code (other than padding) can be found in these, a signature can be generated automatically in many of the cases. Properly integrated with the other tools used at an anti-virus company it should be possible to let Autosig try to generate a signature before a human does. Doing so time can be saved, allowing the human analysts to perform deeper analysis of the more complicated malware.

During the database tests the extent at which signatures were generated did not reach the numbers hoped for, but it is believed that by tuning some of the parameters mentioned above the number of generated signatures can be increased.

Another benefit of a program like Autosig is its ability to clean up in the signature database. Removing redundant signatures and replacing them with working ones is important to keep the database small. Manually doing this is a tedious work and as shown in this chapter it can be done automatically instead.

The test results also show the limitations of the approach used by Autosig. In the case of polymorphic malware it is not possible to generate signatures with such a program.

Autosig Parameters

There are a number of parameters that affects the performance of Autosig. The minimum signature size is such a parameter. During the tests it has turned out that 32 byte might be too small for a signature. Most of the signatures with this size failed to pass the false positives test. By increasing the minimum size it might be possible that less signatures fail the false positives tests.

To this date only four code segments from the list of good signature code have been used for signature generation, even when more were found. The number four was chosen because it is the standard used by the human analysts. By increasing this number signatures more precisely identifying the malware in question could be generated.

Another parameter that affects the signatures generated is the value that starting 2-byte combinations are allowed to have in the statistics. By increasing this value from 0 to 1 the number of allowed starting bytes grows with about 50%, which might lead to more signatures generated. However this could also lead to a deterioration in quality in the signatures generated. The consecutive bytes can also be allowed to have more hits in the statistics to characterize more byte combinations as suitable for signature generation. In both of these cases it has to carefully observed how the false positives rate is affected.

Chapter 7

Conclusion

This Master Thesis shows that it is indeed possible to generate working malware signatures automatically. It shows that, when dealing with malware where many variants of the same family exists, generic signatures for these can be generated fast and accurately. This way the human analysts will have more time to analyze polymorphic, metamorphic or in other ways complicated malware.

The tool Autosig has been implemented and tested. The results show that it is possible to extract a substring suitable for signature generation without performing any analysis of the underlying functionality. Even though the extent at which signatures were generated did not reach the numbers hoped for, it is believed that by fine tuning parameters, such as the minimum signature length, these numbers will raise. By increasing the number of negative signatures it is also anticipated that more working signatures can be generated. To this date only one false negative was caused by the signatures generated by Autosig.

7.1 Future Work

It should be noted that Autosig is a prototype implementation. It shows that automatic signature generation works, but to really benefit from such a tool it has to be properly integrated with the other tools used every day at an anti-virus lab. A number of improvement areas have been identified but not implemented due to time constraints. Some of these are:

Database Connectivity To this date Autosig has generated signatures from local copies of malware samples. The next development step will be to connect directly to the malware database and integrate a search function. This way it will be easier to compare old and new samples of the same malware family and subfamilies. A user could then enter ”Sober” to possibly generate a signature for all Sober samples, or she could pass ”Sober.b” as a parameter in order to just parse the ”b” subfamily of Sober. This step will automate the signature generation process further by removing the steps needed to identify which malware to parse and copying these to the local computer.

Broken Samples Broken files should be removed before Autosig starts parsing the samples. Preferably this should be taken care of before Autosig is executed. This might be implemented as a part of Autosig or as an external tool that cleans up the malware database before execution of Autosig. An external tool has the advantage that it does not slow down signature generation.

Negative Signatures All signatures that cause false positives are saved in the signature database at Ikarus. To this date Autosig has only used the false positives generated by its own signatures. A further step would be to take advantage of all false positive signatures in the signature database too.

It would probably also be profitable to add more code from different exe-packers and other code segments known to be unsuitable for signatures to the negative signatures collection. During the tests it was noted that some signatures were generated from the import tables of the files in question and these signatures often caused false positives. The initial false positive rate could probably be lowered by looking into how these ares can be avoided.

Signature Starting Bytes The signature database is organized according to signature starting bytes. That is, it would be possible to reduce scan time by choosing the same starting bytes as older signatures in the database. This can be achieved by comparing the suggested signatures from Autosig with old ones in the database and, when possible, choose starting bytes that are already present.

Bibliography

[1] Jim Bates. Trojan horse: Aids information introducory diskette version 2.0. *Virus Bulletin*, January:3, 1992.

[2] Dr. Vesselin Bontchev. Current status of the caro malware naming scheme. In *15th Virus Bulletin Conference*, October 2005.

[3] Vesselin Bontchev, Fridrik Skulason, and Dr. Alan Solomon. A new virus naming convention. 1991.

[4] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of ACM*, 20(10):762–772, 1977.

[5] Bsdiff - binary diff/patch utility `www.daemonology.net/bsdiff`, 2006.

[6] Peter G. Capek, David M. Chess, and Steve R. White. Merry christmas: an early network worm. *IEEE Security and Privacy*, 2003.

[7] `CARO` `www.caro.org`, 2006.

[8] David M. Chess and Steve R. White. An undetectable computer virus. *Virus Bulletin Conference*, 2000.

[9] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–18, 1990.

[10] Dr. Frederick B. Cohen. Computer viruses - theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

[11] Dr. Frederick B. Cohen. *A Short Course On Computer Viruses*. John Wiley and Sons, second edition, 1994.

[12] Microsoft Corporation. Microsoft portable executable and common object file format specification. 1999.

[13] Diffbin `http://giraffe-data.com/software`, 2006.

[14] Oren Drori, Nicky Pappo, and Dan Yachan. New malware distribution methods threaten signature-based AV. *Virus Bulletin*, September 2005.

[15] Rich skrenta `www.skrenta.com/cloner`, 2006.

[16] Nick FitzGerald. A virus by any other name: The revised caro naming convention. *AVAR Conference*, 2002.

[17] Sarah Gordon. What is wild? In *The 20th National Information Systems Security Conference*, October 1997.

[18] Kaspersky labs `www.kaspersky.com`, 2006.

[19] Jeffrey O. Kephart and William C. Arnold. Automatic extraction of computer virus signatures. In *Proceedings of the 4th Virus Bulletin International Conference*, pages 178–184, 1994.

[20] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[21] Donald Knuth, James H. Morris, and Jr. Vaughan Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2), 1977.

[22] Donald E. Knuth. *The art of computer programming - Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[23] Thomas Krauss. *Viren Wuermer und trojaner*. Interest Verlag, Kissing, Germany, 2004.

[24] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222, New York, USA, 2005. ACM Press.

[25] Guillermo Mallen-Fullerton. The minimum size of virus identification signatures. In *Proceedings of the Fifth International Computer Virus and Security Conference*, pages 813–817, 1992.

[26] Carey Nachenberg. Behavior blocking: The next step in anti-virus protection. *`http://www.securityfocus.com/infocus/1557`*, March 2002.

[27] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington DC, USA, 2005. IEEE Computer Society.

[28] Sophos `www.sophos.com`, 2006.

[29] Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Computer Communication Review*, 19(1):17–57, 1989.

[30] Symantec `www.symantec.com/avcenter/glossary`, 2006.

[31] Peter Szor. *The art of Computer Virus Research and Defense*. Addison-Wesley, Upper Saddle River, NJ, United States, 2005.

[32] Trend micro `www.trendmicro.com/en/support/pattern594`, 2006.

[33] Virus bulletin `www.virusbtn.com`, 2006.

[34] Virus source code database `www.totallygeek.com/vscdb`, 2006.

[35] Vx heavens `http://vx.netlux.org`, 2006.

[36] Wikipedia `http://wikipedia.org`, 2006.

[37] Wildlist `www.wildlist.org`, 2006.

[38] xdelta `http://xdelta.blogspot.com`, 2006.

Appendix A

W32/Sobig log file

```
[11.08.2006 14:25:31]info  initiating...
[11.08.2006 14:25:31]info  directory 3876 created
[11.08.2006 14:25:31]info  copying files to 3876
[11.08.2006 14:25:31]info  loading scanner
[11.08.2006 14:25:31]info  generating .blks
[11.08.2006 14:25:59]info  removing DOS headers
[11.08.2006 14:25:59]info  sorting .blks
[11.08.2006 14:25:59]info  comparing files
[11.08.2006 14:25:59]info  comparing at same position
[11.08.2006 14:25:59]info  match found
[11.08.2006 14:26:00]info  reading neg_sigs
[11.08.2006 14:26:02]info  reading T3Stat.csv
[11.08.2006 14:26:07]info  finding sig code
[11.08.2006 14:26:08]info  generating signature from file: 85546
                           starting at position: 88
[11.08.2006 14:26:08]info  signature generated
[11.08.2006 14:26:08]info  SIG_EXE_PE 4C98FFFF3BFE7410FF75F48B3
                           D40404100FFD7FF75F0FFD73975087409FF7
                           508FF150C4041006A01 >5 #0xDD5B7F86,44
                           >42 #0x891F1CD6,164 >6 #0xF1793237,48
[11.08.2006 14:26:08]info  signature(s) generated for all samples
[11.08.2006 14:26:08]info  testing signature
[11.08.2006 14:26:09]info  deleting files
[11.08.2006 14:26:09]info  all files in directory 3876 deleted
[11.08.2006 14:26:09]info  directory 3876 deleted
[11.08.2006 14:26:09]info  1 signature(s) generated from a total
                           of 508 samples
[11.08.2006 14:26:09]info  Execution time, in seconds: 38
```

Appendix B

W32/Poplip log file

```
[21.08.2006 17:43:07]info  initiating...
[21.08.2006 17:43:07]info  directory 696 created
[21.08.2006 17:43:07]info  copying files to 3876
[21.08.2006 17:43:31]info  loading scanner
[21.08.2006 17:43:31]info  generating .blks
[21.08.2006 17:45:09]info  removing DOS headers
[21.08.2006 17:45:09]info  sorting .blks
[21.08.2006 17:45:09]info  comparing files
[21.08.2006 17:45:27]info  comparing at same position
[21.08.2006 17:45:27]info  match found
[21.08.2006 17:45:27]info  reading neg_sigs
[21.08.2006 17:45:29]info  reading T3Stat.csv
[21.08.2006 17:45:35]info  finding sig code
[21.08.2006 17:45:35]info  no sig code found
[21.08.2006 17:45:35]info  directory 696 deleted
[21.08.2006 17:45:35]info  0 signature(s) generated from a total
                           of 35 samples
[21.08.2006 17:45:35]info  Execution time, in seconds:  148
```