Chapter 2

Background and Motivation

In this chapter, we provide background information on malicious code. In particular, we present a brief summary of the history of malicious code and answer the question why malware is so prevalent on today's Internet. We also introduce a taxonomy of malware and explain in more detail the characteristics of different incarnations of malicious code. We then discuss previous attempts to fight malware, and in particular, related research that is relevant to the work presented in the thesis. This discussion outlines shortcomings of current malware detection techniques and motivates our search for better and more robust ways to detect malicious code.

2.1 Short History of Malware

The last few years have seen a tremendous increase of malicious code attacks. However, malware is not a new concept and miscreants have been producing evil programs for well over two decades. Figure 2.1 shows a timeline with significant events in the history of malware development. When looking at this timeline in more detail, a number of trends can be observed.

One trend is the increase in sophistication. A virus twenty years ago was a small code fragment that infected executable files and spread via disks that were exchanged between computer users. Current tools are very crafty in their rapid infection and stealth techniques. For example, malware often uses different infection vectors to spread. That is, it is not uncommon that a virus or a worm exploits multiple vulnerabilities in different network services and, additionally, relies on social engineering by sending copies of itself by email, hoping for careless users to click on the attachment. In addition, modern malware often manipulates the operating system kernel to hide its presence and can contain functionality to attack host-based defense systems (e.g., some viruses shut down running anti-virus programs).

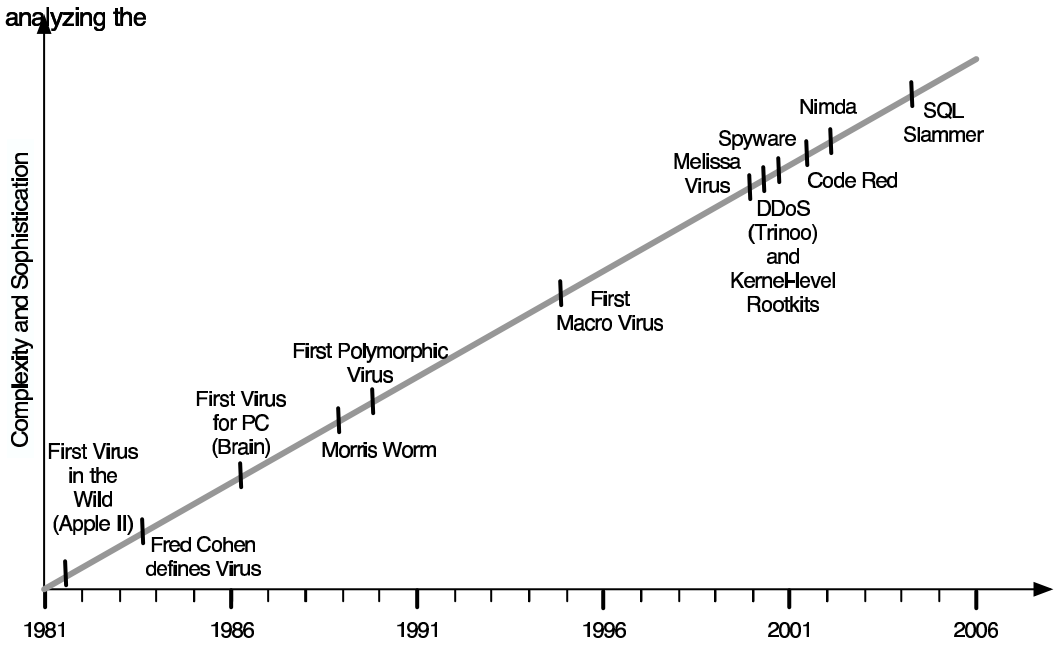

Figure 2.1: 20 years of malicious code.

Another trend is the acceleration of the development of novel techniques. While it took years between significant advances in malware technology in the past, the last few years have seen the emerging of new variations of malicious code at a rapid pace. Many major security threats on the Internet today, such as email viruses, worm epidemics, distributed denial-of-service and botnet tools, spyware, and sophisticated kernel-level rootkits are all a development of the last five years.

The following description of major milestones in malware development are based on the history of malicious code described in [53]:

1981 **First Reported Computer Virus:** At least three separate viruses were discovered in games for the Apple II computer system, although the word virus was not yet applied to this malicious code.

1983 **Formal Definition of Computer Virus:** Fred Cohen [11] defined a computer virus as "a program that can infect other programs by modifying them to include a, possibly evolved, version of itself."

1983 **First Virus for Personal Computer:** The appearance of Brain, the first virus for the IBM personal computer, was an important harbinger of malicious code to come, as the popular DOS and later Windows operating systems would become a primary target for viruses and worms.

1988 **Morris Internet Worm:** Robert Morris released the world's first worm, disabling much of the early Internet.

1990 **First Polymorphic Virus:** To evade anti-virus systems, polymorphic viruses altered their own appearance every time they infected a new file, opening up the frontier of polymorphic (and metamorphic) code that is still being explored in research today.

1995 **First Macro Virus:** Viruses developed until this time were thought of as executable code sequences that infect binary programs. The first macro viruses changed this notion, as they were implemented in the Microsoft Word macro language and infected document files. These techniques soon spread to other macro languages in other programs and broadened the view of malware in general.

1999 **Melissa Virus:** This virus was one of the first to use mail as a means to achieve wide distribution. Written as a macro virus, it also bore characteristics of a worm as it used the network (in this case, the email system) to spread.

1999 **Distributed Denial-of-Service (DDoS):** In late summer of this year, the Tribe Flood Network (TFN) and Trin00 denial of service agents were released. These tools offered an attacker control of hundreds, or even thousands of machines with an installed zombie via a single client machine. With a centralized point of coordination, these distributed agents (often called botnets) could launch devastating denial-of-service attacks or serve as email relays for unsolicited bulk mail (spam).

1999 **Kernel-level Rootkits:** Rootkits are a collection of tools used by an attacker after gaining administrative privileges on a host. Typically, these tools were modified versions of common system administrator tools with the goal of hiding the attacker's presence on the compromised machine. With Knark, the first rootkit was released that was modifying the Linux operating system kernel directly. This allowed an attacker to effectively hide files, processes, and network activity without requiring any modifications to user programs anymore.

2000 **Spyware:** A novel type of software was identified that did not immediately pose a security threat to infected users, but compromised their privacy. This malware variant is monitoring user behavior (e.g., collecting browser habits) and transmitting the gathered information to a third party where it is typically used for market research or targeted advertising purposes. Besides the collection of data, spyware is also often used to display ads directly on the infected machine.

2001 **Code Red Worm:** Code Red was one of the first worms that exploited a vulnerability in a popular network service to spread (a buffer overflow in Microsoft's Internet

Information Service web server). It was very successful, and over 250,000 machines fell victim in less than eight hours.

2001 **Nimda Worm:** The Nimda worm is an example of the new generation of worms that make use of numerous methods for spreading, including Web server buffer overflows, Web browser exploits, Outlook e-mail attacks, and file sharing.

2003 **SQL Slammer Worm:** The SQL Slammer worm is remarkable for its spreading efficiency and small size. The body of the worm fits into a single UDP packet, and the scan engine is fast enough to probe for victims at network speed. This enabled the Slammer worm to infect more than 90% of the vulnerable host population on the Internet is less than eight minutes.

An interesting question to ask is for the reasons why malicious code has become such a prevalent phenomenon in today's networked world. It is true that viruses have always been a nuisance to computer users. However, people had little exposure to malware ten years ago and anti-virus software was run only by a few, cautious users. Currently, almost everybody uses multiple security products such as virus scanners or personal firewalls, and it is a matter of minutes until a freshly installed Windows machine that is connected to the Internet gets compromised by one of the many worms that roam the net.

One major reason, according to [31], is the enormous increase in network connectivity. With the growth of the Internet, both the number of attack vectors and the ease with which an attack can be made increased. More and more computers are connected to the global network, and people, businesses, and governments increasingly rely upon communication such as e-mail or Web pages. As all these systems are connected to the Internet, they become vulnerable to attacks from distant sources. In particular, it is no longer necessary for an attacker to obtain physical access to a system to install or propagate malicious code. This lowers the barrier, as one can simply attack systems from the comfort of one's living room. In addition, attacks can be automated and a single person can comprise hundreds, or even thousands, of machines within seconds. This lead to the emerge of denial-of-service attacks where a botnet comprised of a large number of previously compromised hosts simultaneously flooded a victim site.

A second problem is the increasing size and complexity of modern information systems. A desktop system running Windows and associated applications depends upon the proper functioning of the kernel as well as the applications to ensure that malicious code cannot corrupt the system. However, Windows itself is a complex monolith that consists of tens of millions of lines of source code, and applications are becoming equally large. When systems become this large, bugs cannot be avoided. This problem is exacerbated by the use of unsafe programming languages (such as C or C++) that do not protect against simple memory corruption attacks (such as buffer overflows). Unfortunately, even if the

systems were bug free, improper configuration can open the door to malicious code. In addition to providing more avenues for attack, complex systems make it easier to hide or mask malicious code. In theory, one could analyze and prove that a small program was free of malicious code, but this task is impossible for real-world systems.

A third problem is the degree to which systems have become extensible. On one hand, this problem refers to the fact that the operating system or applications can be dynamically enhanced with new functionality. For example, some operating systems support extensibility through dynamically-loadable device drivers and modules, while a web browser can be extended by a new plug-in. This lead to the emerge of malicious code such as kernel-level rootkits that install themselves as operating system modules, or malware that exploits security vulnerabilities in the ActiveX extension system of Microsoft Internet Explorer. On the other hand, extensibility refers to the possibility to embed code snippets or macros into data objects. Word documents support a simple macro language to automate repetitive tasks and JavaScript can be embedded into web pages to improve their presentation or ease navigation. Extensible systems are attractive because they provide flexible interfaces that can be adapted through new components. Unfortunately, it is hard to prevent malicious code from slipping in as an unwanted extension. This is demonstrated by attacks such as cross-site scripting (XSS), where malicious JavaScript is injected into web pages, or Word macro viruses. An example for problems due to extensible systems, the Melissa virus took advantage of the scripting extensions of Microsoft's Outlook e-mail client to propagate itself. The virus was coded as a script contained in what appeared to users as an innocuous mail message. When the message was opened, the script was executed, and proceeded to obtain email addresses from the user's contacts list before sending copies of itself to those addresses.

2.2 Malware Taxonomy

Malicious code is a rich and diverse field, and a number of different approaches to classify malware exist. In this thesis, we propose a taxonomy of malware that focuses on two important properties: (1) the ability of a piece of code to spread autonomously, and (2) its ability to run independent of a host program.

Of course, our taxonomy provides only a coarse guideline to classify malware. In particular, we do not distinguish between the different mechanisms that can be used to spread. Some taxonomies [31] introduce a fine-grain distinction between malware that spreads by exploiting vulnerabilities in remote services, programs that send copies of themselves by email, or virus code that is spread via infected files exchanged by file sharing or on removable media such as floppy disks. In our case, we are only interested in the mere fact that a piece of malware is capable of autonomously creating copies of itself.

Another area that we do not consider for our taxonomy is the mode of operation of malicious code, or, in other words, the damage that it is supposed to inflict. Unfortunately, the purpose of malicious code is only limited by the imagination of the miscreant who created it, and many malware programs implement routines to cause damage in different ways. Sometimes, malicious code even spreads without effecting its host in a negative fashion. More often, however, programs or data on infected computer systems are destroyed. Other functionality includes the installation of a backdoor to allow an intruder easy access at later times, the flooding of a victim host with useless traffic to carry out a denial-of-service attack, or the installation of a simple mail server to distribute spam messages.

Our simple taxonomy, including examples for each class, can be found in Figure 2.2. The malware instances introduced in this figure are discussed in more detail below.

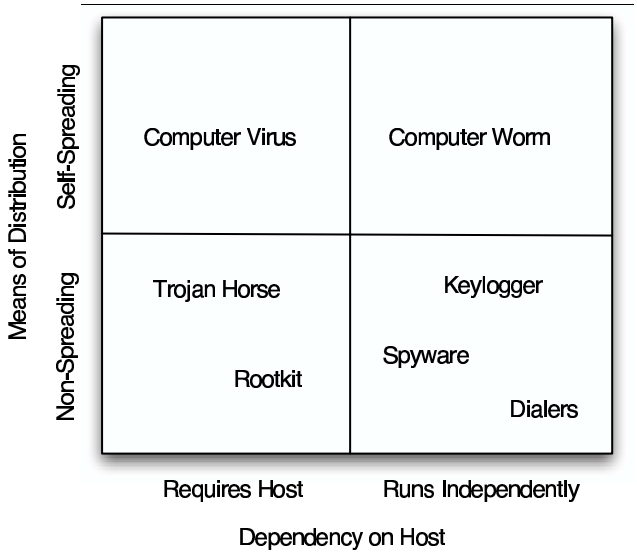

Figure 2.2: Taxonomy of malicious code.

Computer Virus: As mentioned previously, the first definition [11] of a computer virus was given by Fred Cohen in 1984. He classified a virus as "a program that can infect other programs by modifying them to include a, possibly evolved, version of itself." This definition characterizes a virus as a program that spreads and that requires a host program where it can include a copy of itself. Interestingly, the definition already alluded to the possibility of code evolution. While it is common nowadays that viruses make use of polymorphic or metamorphic techniques to alter their appearance in order to evade pattern-(or signature-)based detection, this concept was not yet introduced at the time Cohen made his statement. The aforementioned definition is still accepted today, although it had to be modified slightly to take into account viruses that do not directly modify their host programs. Instead, so called companion viruses alter the program's environment and place

themselves in front of the victim's program on the execution path, of course using the same name as the victim application. By doing so, the operating system always starts the virus program instead of the actual application when attempting to launch the program. Thus, an updated definition of a virus is given by Peter Szor [58] as "a program that recursively and explicitly copies a possibly evolved version of itself."

Computer Worm: A computer worm is closely related to a computer virus, as both types of malware have the ability to spread autonomously. The main difference between both types is the fact that a worm is a stand-alone application while a virus requires a host program that it can infect. Also, although not strictly necessary, computer worms typically spread via the network. This is either done by exploiting a vulnerability in a network service or by sending copies of the worm as email attachments. In both cases, whenever a worm successfully compromises a host, it starts to actively search for new victims. This can be done by scanning the network for the occurrence of remote hosts that expose a vulnerable service or by scanning the local machine for email addresses.

Trojan Horse: A Trojan horse is a computer program that is hidden inside another program that serves a useful purpose. For example, a program that appears to be a game or image file but in reality performs some other malicious function would be classified as a Trojan horse. The term Trojan horse comes from the mythical deceit of war used by the Greeks against Troy. According to our taxonomy, Trojan horses are classified as malware that requires a host because they are embedded into another application that performs something useful. In contrast to computer viruses and worms, Trojan horses do not spread by themselves. Instead, they must be installed explicitly on a machine by an attacker.

A widespread type of Trojan horse are rootkits. Rootkits are code introduced into system administration tools with the purpose of hiding the presence of an attacker on the system. This is typically realized by suppressing that part of the output of these programs that would reveal information about the intruder's processes, files, or network connections. The most insidious type of rootkits are kernel-level rootkits. This instance of malicious code is implemented as modules that integrate themselves into the operating system. That is, kernel-level rootkits are Trojan horses that use the operating system as host.

Spyware: Spyware is a term used for programs that monitor the behavior of users and steal private information, such as keystrokes or browsing patterns. This is different from other types of malware, such as viruses and worms, which typically aim to cause damage or to spread to other systems as quickly as possible. Spyware is malware that is typically realized as stand-alone applications that must be installed on victim's machine. Interestingly, spyware often comes bundled with free software that explicitly states that spyware is installed on a user's machine. However, this information is hidden in end-user license

agreements that are frequently dozens or even hundreds of pages long. Thus, a spyware producer can be almost certain that nobody reads these license agreements and simply agrees to have the malware installed.

The information collected by a spyware program is sent back to the spyware distributors and used as a basis for targeted advertisement (e.g., pop-up ads) or marketing analysis. Spyware programs can also "hijack" a user's browser and direct the unsuspecting user to web sites of the spyware's choosing. Finally, in addition to the violation of users' privacy, spyware programs are also responsible for the degradation of system performance because they are often poorly coded.

A key-logger is a common incarnation of spyware that focuses on the recording of the keys that a user types. Other common spyware instances are modules for the Internet Explorer web browser, where they have an excellent vantage point for observing a user's browsing behavior.

Dialer: A dialer is a computer program which creates a connection to the Internet or another computer network over the analog telephone or ISDN network. The concept is to make money for the people that develop the dialer by having the victims connect to the Internet via premium-phone numbers (which are, of course, controlled by those that distribute the dialer malware). While still in use today, the increasing availability of broadband Internet and the concurrent decrease of people that use dial-up connections reduces the number of possible victims and thus, limits dialer fraud.

2.3 Malicious Code Evolution

Malicious code is constantly evolving as malware authors aim to evade the state-of-the-art detection engines and virus scanners. In this section, we shed light on the evolution of malicious code and attempts of the anti-virus industry to keep up in the arms race against malware authors.

The first malware instances were simple viruses that appended their invariant body to each file that was infected and redirected the program entry point to the start of the virus routine. To detect such viruses, first-generation scanners were equipped with a set of byte-level signatures that exactly describe a characteristic part of each known piece of malicious code. As a way to counter detection schemes based on exact byte signatures, viruses made soon use of encryption to hide the actual virus body. That is, the actual virus body was decrypted and prepended by a decryption routine. Some viruses were even shipped with a number of different decryptors to make their identification more difficult. Of course, virus writers quickly realized that the detection of their creations remains trivial when the decryption routine does not change between different virus generations.

In an attempt to obfuscate the presence of a decryption routine, simple metamorphic techniques were introduced. These techniques were not aimed at morphing the malicious code *per se*, but to generate as many variations of the decryption loop as possible. The first virus that used a mechanism called *dead code insertion* to camouflage the decryption routine was named `1260`. For this virus, the author created a skeleton of fixed instructions that were necessary for the correct functionality of the decryptor. In between these code islands, the virus would randomly insert junk different instructions every time it infects a new file. These junk instructions were selected from a list of operations that were known not to interfere with the actual decryption functionality. An example of such an operation was the x86 `nop` instructions, which literally performs no operation and keeps the state of the process untouched. Other instructions were arithmetic operations that performed useless calculations on registers that were not used by the virus code. The purpose of mixing random junk instruction with actual code was to change the layout of the virus in every generation so that no signature would match more than a few instances of a virus.

Other approaches to force different code layouts were quickly developed. One such technique, called *register reassignment*, makes use of the fact that a processor possesses a number of general purpose registers that can be used interchangeably for computations. Similar to a compiler back-end during code generation, a virus that employs register reassignment selects a certain register for a sequence of interrelated instructions that perform a particular calculation. In each virus generation, a different register can be chosen. Selecting a different register for an operation causes a slight change in the encoding of the corresponding instruction. This leads to changes in the code layout that can evade precise signatures. However, signatures that employ wildcards can still be used to detect such worms. A detection engine supports wildcards when it is not necessary to precisely specify the exact value of consecutive bytes in a signature. Instead, it is possible to leave certain bytes in a sequence undefined. As an example, consider Figure 2.3 that shows the disassembly and code layout of two different generations of the `W95/Regswap` virus, discovered in 1998. It can be seen that most of the code (shown in bold face) remains unchanged. Thus, a signature that specifies a sequence of bytes with a few unspecified locations in between can still be written to characterize this virus.

Yet another method to morph the syntactic code layout of malware is called *code transposition*. A virus that uses code transposition in its simplest form has a decryption routine that is divided into a number of modules, or subroutines. Each module itself does not change between different virus generations, but their order can be rearranged. This allows a virus to generate $n!$ different instances of itself, where n is the number of modules. Typically, modules execute in the same order in each virus generation. To this end, the last instruction of each module is an unconditional jumps to the next one in the sequences. Of course, when modules are shuffled, the targets of these jumps need to be updated to reflect the changed order. One of the first viruses that used code transposition on the

```
5A                  pop %edx
BF04000000          mov %edi,$0x04
8BF5                mov %esi,%ebp
B80C000000          mov %eax,$0x00
81C288000000        add %edx,$0x88
8B1A                mov %ebx,(%edx)
899C8618110000      mov 0x1118(%esi,%eax,4),%ebx
```

(a) First virus generation

```
58                  pop %eax
BB04000000          mov %ebi,$0x04
8BD5                mov %edx,%ebp
BF0C000000          mov %edi,$0x00
81C088000000        add %eax,$0x88
8B30                mov %esi,(%eax)
89B4BA18110000      mov 0x1118(%edx,%edi,4),%ebx
```

(b) Second virus generation

Figure 2.3: Two generations of `W95/Regswap`.

level of modules was `BadBoy`, a DOS virus with eight different subroutines that could be rearranged for a total of about 40,000 different variants. A more sophisticated version of code transposition does not move complete blocks but individual instructions. To this end, the virus author defines independent instructions that can be executed in an arbitrary order relative to each other.

Finally, some viruses make use of *command substitution*. For tasks such as setting the content of a register to zero, the rich Intel x86 instruction set provides a number of semantically equivalent instructions (e.g., using `xor %reg,%reg` or `sub %reg,%reg`). Typically, command substitution is implemented with the help of command tables that hold different code sequences for a particular operation. Whenever the virus creates a new instance of itself, it consults this table to randomly select one of the appropriate code sequences for each tasks that it needs to perform. Because different (but semantically equivalent) sequences are chosen in different generations, two virus samples have a different layout with a high probability (of course, it might happen by accident that two virus instances are assembled from the same code sequences, but the likelihood for this to occur is low).

Malicious code that uses an encrypted body with a metamorphic decryption engine constitutes a massive problem for first-generation malware detectors. The reason is that such malware appears different in every generation and there is no longer a byte signature that can be used to match the malicious code. The reaction of the anti-virus vendors was the development of second-generation scanners. These scanners exploit the fact that no matter how complicated and involved the decryption routine is, the virus body is still invariant *after* is has been decrypted. To decrypt a virus, the advanced scanners make use of code emulation. More precisely, these scanners contain a simple virtual processor that

can execute a program in a protected sandbox. When the virus is executed, its decryption routine makes sure that the actual virus body is unpacked. At this point, the scanner can again make use of simple signatures. As mentioned previously, the reason is that after the execution of a complex decryption routine, the actual virus body that is dynamically decrypted in memory remains fixed for (and unique to) a certain malware sample.

Given the power of metamorphic code transformations, it is obvious that these techniques are not limited to their application to the decryption routine. Indeed, it would be much better from the point of view of a malware author if the complete virus is metamorphic. That is, there is no encryption used anymore, as the body of the virus itself has in a different layout in every generation. This also has the advantage that second-generation scanners fail, because there is no decryption process and no invariant virus body that can be detected. Although a complete metamorphic virus is appealing to a malware author, the technical difficulty in writing a proper functioning piece of metamorphic code has so far deterred most developers. Nevertheless, a few such viruses were spotted in the wild. Among the most infamous ones are `W95/Zmist` and `W95/Smile`. In addition to strong metamorphic features, using all the techniques outlined above, each virus exhibits a special feature that makes it unique.

`W95/Zmist` [58] supports code integration. This means that the virus does not simply append a copy of itself to a file, but instead moves instructions of the original application to make room for the virus code. In other words, the virus body is merged with the application code. To make room for the virus' new instructions, the existing code section is expanded and existing instructions are shifted. This has an important consequence: Instructions that are targets of jump or call operations are relocated. As a result, the operands of the corresponding jump and call instructions need to be updated to point to these new addresses. Note that this also effects relative jumps, which do not specify a complete target address, but only an offset relative to the current address. In some cases, relocating the target instruction even moves this instruction out of the range of a short jump. If this happens, the short jump has to be converted into an appropriate long jump. To perform the tasks outlined above, the host application has to be accurately disassembled. This is not an easy task, especially when targeting x86 binaries on Microsoft Windows. The reasons are the variable instruction length of the Intel x86 processor and the fact that code and data are mixed in the code section. Despite these difficulties, the virus is typically successful and most infected applications still run after they have been modified by `W95/Zmist`.

`W95/Smile` [58] in interesting because it uses a processor-independent, intermediate code to generate its metamorphic variants. More precisely, when the virus spreads, it first translates the viral code into an intermediate language that is independent of the actual CPU the virus runs on. Then, metamorphic obfuscation transformations are applied to this intermediate virus representation. Finally, the code is compiled back into the native form.

This process demonstrates that the virus in fact contains a simple binary transformation engine. The use of an intermediate code representation would also allow the virus to infect programs that are written for a different platform than x86, although such a behavior has not been observed yet.

Even when employing code emulation, second-generation scanners cannot use their standard approach to detect metamorphic worms such as `W95/Zmist` or `W95/Smile`. To provide some level of protection against such malware, scanners resort to "algorithmic detection schemes." In practice, algorithmic detection is just an euphemistic term for heuristics that are developed by anti-virus vendors to detect a particular instance of malware. For example, a virus often tags an executable by writing a special character to a particular location in the executable header or into the code segment. These tags are used to identify previously infected files to prevent the virus from infecting the same program multiple times. A scanner can then check for the presence of such tags to find infected files. Other heuristics recognize executables that have unusual code sections or unusual header entries. Yet others check for suspicious hard-coded addresses that are utilized by a virus. Unfortunately, algorithmic detection schemes can be easily evaded by a malware author, thus making frequent updates to the virus database necessary whenever a new variant of a previously unknown piece of malicious code is detected in the wild. As put by Peter Szor [58], chief malware analyst at Symantec: "Heuristic systems can only reduce the problem against masses of viruses. The evolution of metamorphic viruses is one of the greatest challenges of this decade."

2.4 Previous Research

The first academic papers to deal with the problem of malicious code were presented by Christodorescu et al. [5, 6]. In these papers, the authors demonstrated the ease of evading state-of-the-art commercial virus scanners. In their experiments, all systems failed to detect a mutated virus after applying syntactic transformations to the virus body (e.g., reordering instructions, renaming registers). This is not surprising, though, as the anti-virus products simply did not have suitable signatures to characterize the virus instances that resulted from the code transformations. In fact, these results mostly confirm that commercial scanners cannot detect novel virus variants because they use malware descriptions that are very specific to particular virus instances. In [5], the authors also present a static analysis approach that can be used to revert certain code transformations. That is, the authors presented a tool, which is based on model checking, that recognizes that a code sequence is identical to a given virus specification, even when this code sequence was modified by code reordering, register reassignment, or junk insertion. The system still lacks the ability to detect novel malware instances, however, because a precise specification is necessary to

characterize a virus. Nevertheless, the approach offers the benefit that the tool cannot be confused by simple code transformations.

The first paper that argues for more a general description of malware was presented by the author of this thesis in [26]. The key idea presented in this paper, and also the insight which constitutes the base of the thesis, is the fact that it should be possible to describe the behavior of malware on a higher level of abstraction, at least for certain classes of malicious code. That is, instead of providing one specific description for each different piece of malware, it would be desirable to classify a whole class of malicious code by the behavior it exhibits.

In [26], we focused on the static detection of kernel-level rootkits. To distinguish the behavior of rootkits from legitimate operating system modules, we defined malicious behavior as write accesses to forbidden regions in the kernel address space. This provided us, independent of the actual implementation of the kernel rootkit, with a high-level, behavioral specification that has the power to capture the characteristics of *all* kernel rootkits. This has two important implications. First, our analysis is immune to transformations that merely modify the syntactic appearance of the malware. Because metamorphic transformations do not alter the semantics of the code, our detection approach is robust to such obfuscation attempts. The second implication is that our system has the power to detect previously unknown malware instances. Because we specify a behavior that is characteristic for a *whole class* of malware, all representatives that exhibit this behavior are classified as unwanted. Indeed, our results demonstrated that we were able to detect previously unknown rootkits with no false alarms.

The idea of behavioral characterization of malicious code was later adopted in [7] and [22]. In [7], the authors focus on the decryption loop to capture the general behavior of polymorphic viruses. The authors state that such malware can be described by a behavioral specification that defines "(1) a loop that processes data from a source memory area and writes data to a destination memory area, and (2) a jump that targets the destination area." The authors further introduce a number of sophisticated static analysis techniques that they employ to search for suspicious behavior in binaries. In [22], the authors focus on the spreading mechanism of worms. In particular, the authors classify a program as virus when it creates a copy of itself. To this end, model checking is used to analyze a program's control flow graph for the occurrence of suspicious code sequence. Such a suspicious code sequence is defined as a call to the `GetModuleFileNameA` Windows API function, followed by an invocation of the `CopyFileA` function. Of course, the specification requires that the return value of `GetModuleFileNameA`, which yields the programs name, is used as a parameter to the `CopyFileA` routine.

All approaches described so far use static analysis techniques to locate suspicious code sequences in binary executables. This technique requires that the malware detector can obtain an accurate disassembly of the program under analysis. In [29], Linn and Debray

introduced novel obfuscation techniques that exploit the fact that the Intel x86 instruction set architecture contains variable length instructions that can start at arbitrary memory address. By inserting padding bytes at locations that cannot be reached during run-time, disassemblers can be confused to misinterpret large parts of the binary. To address this problem, we presented a robust disassembler approach in [27].

Besides static techniques, dynamic analysis techniques play an important role both in the manual and automatic analysis of malicious code. Previously, we have already pointed out that second-generation virus scanners make use of code emulation techniques to bypass the obfuscated decryption routines of polymorphic malware. A problem is that malicious code is often equipped with detection routines that check for the presence of a virtual machine or a simulated OS environment [46, 48]. When such an environment is detected, the malware modifies its behavior and the analysis delivers incorrect results. Malware also checks for software (and even hardware) breakpoints to detect if the program is run in a debugger [59]. This requires that a dynamic analysis environment is invisible to the malware that is executed.

We have developed a dynamic analysis engine that runs malicious code in an emulated operating system environment while monitoring its (security-relevant) actions. In particular, we record the Windows native system calls and Windows API functions that the program invokes. One important feature of our system is that it does not modify the program that it executes (e.g., through API call hooking or breakpoints), making it more difficult to detect by malicious code. Also, our tool runs binaries in an unmodified Windows environment, which leads to excellent emulation accuracy. Interestingly, a system with quite similar features and design has been concurrently developed and will be presented in [60].

2.5 Summary

In this chapter, we provided a brief summary of the history of malicious code and introduced major classes of malware. We also outlined the evolution of malicious code, in particular various metamorphic techniques used by malware authors to evade detection by commercial virus scanners. We also pointed out the two main limitations of current systems for malicious code detection, namely their inability to identify previously unseen malware and their sensitivity to code obfuscation. Finally, we suggested to solve the aforementioned problems by characterizing malware at a higher level of abstraction, using behavioral and structural signatures. The details of this solution and its application to the detection of kernel-rootkits and spyware form the core of the following chapter.