SCIENTIFIC and TECHNOLOGICAL COOPERATION

between

RTD ORGANISATIONS in GREECE

and RTD ORGANISATIONS in U.S.A, CANADA, AUSTRALIA, NEW ZEALAND, JAPAN, SOUTH KOREA, TAIWAN, MALAISIA and SINGAPORE

HELLENIC REPUBLIC

MINISTRY OF DEVELOPMENT

GENERAL SECRETARIAT FOR RESEARCH & TECHNOLOGY

International S & T Cooperation Directorate, Bilateral Relations Division

Project MILTIADES: **M**ult**I-L**ayer **T**echn**I**ques for **A**ttack **DE**tection **S**ystems

Deliverable 2.1: System Design

This document presents the detailed design of the two main defense approaches under development within the project MILTIADES, namely the *IP address space randomization* proactive defense technique against hit-list worms, and the *emulation-based polymorphic attack detection* technique.

Due Delivery Date	31/3/2007
Actual Delivery Date	2/5/2007

Project Partners

FORTH-ICS	Project Leader	Greece
Columbia University	Project Leader	US
Virtual Trip	Co-operating organisation	Greece

Table of Contents

1. Introduction

As a response to the ever increasing number of automated Internet attacks, as well as to the increasing levels of attack sophistication, the project MILTIADES explores novel attack detection and defense approaches against modern cyber-attacks. Specifically, our contributions revolve around the following two research directions: i) IP address space randomization, and ii) emulation-based polymorphic attack detection.

1.1 IP address space randomization

To evade detection during their outbreak, and generate as little traffic as possible, sophisticated worms gather information about their targets several weeks before they launch their attack. During those weeks they probe a very large number of IP addresses, if possible all 4 billions of them available in the Internet today (IP version 4), in order to find those hosts which are vulnerable to the planned attack. All vulnerable hosts found, are included in a special list of targets, which is frequently referred to as the "hit-list". Instead of attacking computers at random during the outbreak, as naive worms do, sophisticated hit-list-based worms only attack computers included in the hitlist and therefore (1) they generate the minimum traffic possible, and (2) they do not generate any unsuccessful (TCP) Internet connections. Therefore, hit-list-based worms propagate at the maximum possible speed, evading their timely detection by worm detection systems.

To slow down the rate of their spread, and if possible neutralize those hit-list worms, we propose to conduct research towards mechanisms which will make the contents of the hit list stale. Armed with a stale hit list, a worm will not be able to successfully infect hosts, but it will also make several unsuccessful attempts to compromise non-vulnerable computers, resulting in several unsuccessful TCP connections. These unsuccessful connections will probably be more visible to firewalls and intrusion detection systems, which will quickly take notice of the spreading worm. To put it simply: a stale hit-list will slow down the spreading worm, and make it visible to firewalls and Intrusion Detection Systems.

1.2 Emulation-based polymorphic attack detection

To protect themselves against malicious intruders, organizations usually employ Network-level Intrusion Detection Systems (NIDSes) in their gateways to the Internet. NIDSes, inspect all incoming traffic against a preloaded set of known attack signatures (i.e., attack rules) in order to see whether any network packet(s) match any of the attack signatures. As soon as the NIDSes find network packets which match any one of the attack signatures, they log the offending packets and alert the system administrators of the intrusion attempt.

Although IDSes have been successfully used to identify and prevent traditional attacks, they are getting increasingly less effective when faced with the next generation of polymorphic and metamorphic worms for several reasons. First, traditional Intrusion Detection Systems operate using predefined attack signatures, which means that they cannot

detect previously unknown (“zero day”) attacks for which no signature exists. Second, as organizations start deploying state-of-the-art detection technology, attackers are likely to react by employing advanced evasion techniques, such as polymorphism and metamorphism, to defeat these defenses.

In contrast to previous work, in this project we explore the approach of *emulation-based polymorphic attack detection*, a novel approach for the detection of previously unknown polymorphic attacks, which is based on the actual execution of attack data on a CPU emulator. Our approach does not rely on any exploit or vulnerability specific signatures, which allows the detection of previously unknown attacks. The main principle of the approach is the use of a generic heuristic that matches the runtime behavior of polymorphic shellcodes. At the same time, the actual execution of the attack code on a CPU emulator makes it robust to evasion techniques such as highly obfuscated or self-modifying code. Furthermore, each shellcode is detected separately, which makes it effective against targeted attacks.

2. IP Address Space Randomization

Worms and viruses are widely regarded to be one of the major security threats facing the Internet today. Incidents such as Code Red [CODERED01, MOORE02] and Slammer [SLAMMER03] have clearly demonstrated that worms can infect tens of thousands of computers in less than half an hour, a timescale where human intervention is unlikely to be feasible. More recent research studies have estimated that worms can infect as many as a million hosts in less than two seconds [STANIFORD04, STANIFORD02, WEAVER04]. Unlike most of the currently known worms that find their victims by targeting random IP addresses in search for vulnerable hosts, these extremely fast worms rely on *hitlists*, precomputed lists of vulnerable targets, in order to spread efficiently.

We have considered the question of whether it is possible to defend against hitlist worms. We first examine strategies for building hitlists and how effective these strategies can be. We observe that hitlists tend to *decay* naturally for various reasons, as hosts get disconnected or change addresses, and applications are started and shut down. A rapidly decaying hitlist is likely to decrease the spread rate of a worm. It may also increase the number of unsuccessful connections it initiates, and may thus increase exposure of the worm to scan-detection methods. Starting with this observation about hitlist decay, it is natural to ask if it is possible to *intentionally* induce hitlist decay, and we examine the possibility of achieving this through *network address space randomization* (NASR). This technique is inspired by instruction address randomization that has been proposed to protect against code injection attacks at the compiler level[GAURAV03]. It is also similar in principle to the "IP hopping" mechanism presented in [ATIG03], whose goal is to confuse targeted attacks. We apply the same basic idea to the specific context of defending against hitlist worms. In its simplest form, network address space randomization can be provided by adapting dynamic IP address allocation services such as DHCP to *force* more frequent address changes. This simple approach may be able to protect enabled networks against hitlist worms, and, if deployed at a large enough scale, may be able to significantly hamper their spread.

2.1 Hitlists

Instead of attempting to infect random targets, a worm could first determine a large vulnerable population before it starts spreading. The worm creator can assemble a list of potentially vulnerable machines prior to releasing the worm, for example, through a slow port scan. The list of known vulnerable hosts is called a hitlist. Using hitlists, worms do not need to waste time scanning for potential targets during the time of the attack, and will not generate as many unsuccessful
connections as when scanning randomly. This allows them to spread much faster, and it also makes them less visible to scan-based worm detection tools. A hitlist can be either a collection of IP addresses, a set of DNS names or a set of Distributed Hash Table identities (for infecting DHT systems irrelevantly of the network infrastructure).

There are many ways for building hitlists. Random scanning can be used to compile a list of IP addresses that respond to active probing. Since the addresses will not be used im-

mediately, the worm author can use so-called stealth, low rate, scanning techniques to make the scan pass unnoticed. On the other hand, if the duration of the low-rate scanning phase is very long, some IP addresses will eventually expire. Hitlists of Web servers can be assembled by sending queries to search engines and by harvesting Web server names off the replies. Similar single-word queries can also be sent to DNS servers in order to validate web server names and find their IP addresses. Interestingly enough, these types of scans can be used to easily create large lists of web servers, and are very likely to go unnoticed.

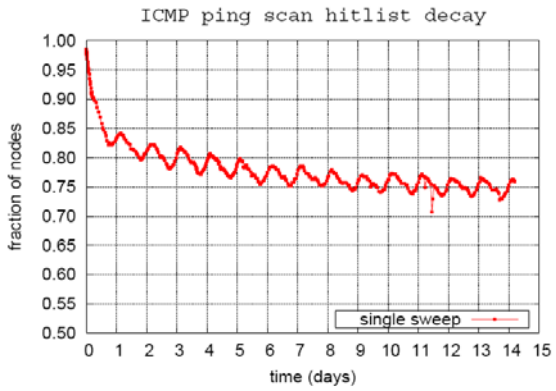

Figure 1: Decay of addresses using random scanning

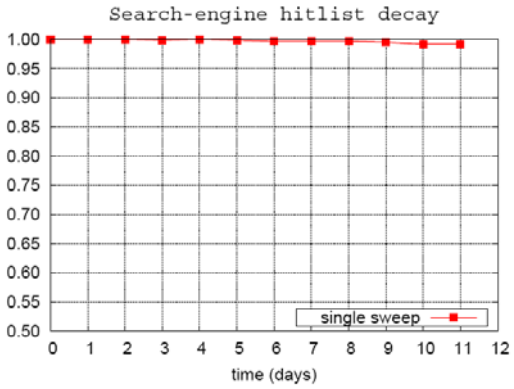

Figure 2: Decay of addresses harvested by querying a popular web search engine

To explore the design space of network address space randomization we first need to consider some basic hitlist characteristics, such as the speed at which a hitlist can be constructed, the rate at which addresses already change (without any form of randomization), and how address space is allocated and utilized. We perform measurements on the Internet to obtain a clearer picture of these characteristics.

2.1.1 Random scanning

We determine the effectiveness of random scanning for building hitlists. We first generate a list of all /16 prefixes that have a valid entry with the *whois* service, in order to increase scan success rates and avoid reserved address space. We then probe random targets within those prefixes using ICMP ECHO messages. Using this approach, we generated a hitlist of 20,000 addresses. Given this hitlist, we probe each target in the hitlist once every hour for a period of two weeks. Every probe consists of four ICMP ECHO messages spaced out over the one-hour run in order to reduce the probability of accidentally declaring an entry stale because of short-term congestion or connectivity problems.

The results of the ICMP ECHO experiment are shown in Figure 1. We observe that the hitlist decays rapidly during the first day, and continues to decay, albeit very slowly, over the rest of the two-week run. Overall, the decay of the hitlist slows down over time, reaching an almost stable level of 75% of hitlist nodes reachable.

2.1.2 Search-engine harvesting

Querying a popular search engine for *the* or similar keywords returns hundreds of millions of results. Retrieving a thousand results gave 612 unique alive hosts and 30 dead hosts. Most search engines restrict the number of results that can be retrieved, but the attacker can use multiple keywords, either randomly generated or taken from a dictionary. Figure 2 shows the decay of the hitlist created using the search engine results. We observe that, compared to the other address sources, the search engine results are very stable. This was expected, since web servers have to be online and use stable addresses. It does not mean, however, that addresses retrieved through search engines are better suited for attackers. Depending on the vulnerability at hand, unprotected, client PCs, such as those returned by crawling peer-to-peer networks may be preferred.

2.2 Network-address space randomization

The goal of network address space randomization (NASR) is to force hosts to change their IP addresses frequently enough so that the information gathered in hitlists is rendered stale by the time the hitlist-based worm is unleashed. To illustrate the basic idea more formally, consider an abstract system model with an address space R= {1,2,…,n}, a set of hosts H = $\{h_1,\ldots h_m\}$ where m<n, and a function A that maps all hosts h_k to addresses $A(h_k) = r$, where r belongs to R. Assume that at time t_a, the attacker can (instantly) generate a hitlist X containing the addresses of hosts that are live and vulnerable at that time. If the attack is started at time t_x and all hosts in X are still live and vulnerable and have the same address as at time t_a then the worm can very quickly infect |X| hosts.

In a system implementing NASR, consider that at time tb where $t_a < t_b < t_x$, all hosts are assigned a new address from R. Thus, at the time of the attack the probability that a hitlist entry x_k still corresponds to a live host is p = m/n and thus the attacker will be able to infect (m/n)|X| hosts. Besides reducing the number of successfully infected nodes in the hitlist, the attack will also result in a fraction 1-(m/n) of all attempts failing (which may be detectable using existing techniques). In this simple model, the density m/n of the address space seems to be a crucial factor in determining the effectiveness of NASR. So far we have assumed a homogeneous set of nodes, all with the same vulnerability and probability of getting infected. If only a subset of the host population is vulnerable to a certain type of attack, then the effectiveness of NASR in reducing the fraction of infected hitlist nodes and the number of failed attempts is proportionally higher.

2.2.1 Practical considerations

The model we presented illustrates the basic intuition of how NASR can affect a hitlist worm. Mapping the idea to the reality of existing networks requires us to look into several practical issues. First, random assignment of an address from a global IP address space pool is not practical for several reasons: (i) it would explode the size of routing tables, the number of routing updates, and the frequency of recomputing routes. (ii) it would result in tremendous administrative overhead for reconfiguring mechanisms that make address-based decisions, such as those based on access lists, and (iii) it requires global coordination for being implemented and is thus less practical. The difficulty of

implementing NASR decreases as we restrict its scope to more local regions. Each AS could implement AS- or prefix-level NASR, but this would still create administrative difficulties with interior routing and access lists. It seems that a reasonable strategy would be to provide NASR at the subnet-level, although this does not completely remove the problems outlined above. For instance, access lists would need to be reconfigured to operate on DNS names and DNS would need to be dynamically updated when hosts change addresses. Second, some nodes cannot change addresses and those that may not be able to do so as frequently as we would want. The reason for this is that addresses have first-class transport- and application-level semantics. For instance, DNS server addresses are usually hardcoded in system configurations. Even for DHCP-configured hosts, changing the address of a DNS server would require synchronizing the lease durations so that the DNS server can change its address at exactly the same time when *all* hosts refresh their DHCP leases. While technically feasible, this seems too complex to implement and such complexity should rather be avoided. Similar constraints hold for routers. Generally, all active TCP connections on a host that changes its address would be killed, unless connection migration techniques such as [WILLIAMSON02, BARATTO04] are used. Such techniques are not widely deployed yet and it is unrealistic to expect that they will be deployed soon enough to be usable for the purposes of NASR.

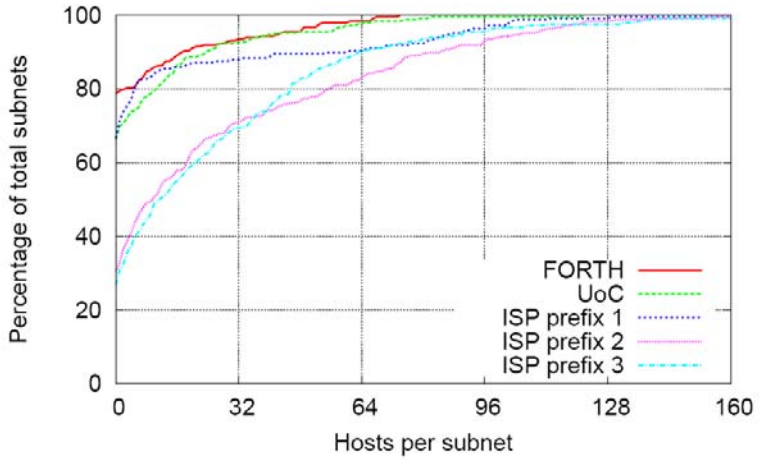

Figure 3 Subnet address space utilization

Fortunately, many applications are designed to deal with occasional connectivity loss by automatically reconnecting and recovering from failure. For such applications, we can assume that infrequent address changes can be tolerated. Examples of these applications are many P2P clients, like Kazaa and DirectConnect, Windows/SAMBA sharing (when names are used), messengers, chat clients, etc. However, tolerance does not always come for free: frequent address changes may result in churn in DHT-based applications, and would generally have the side-effect of increasing stale state in other distributed applications, including P2P indexing and Gnutella-like host caches. Finally, some applications are even less tolerant to failures. For instance, NFS clients often hang when the server is lost, and do not transparently re-resolve the NFS server address from DNS before reconnecting. There exist ways to make systems more robust to address changes. Rocks [BARATTO04] is one solution providing reliable sockets for protecting applications sensitive to IP address changes. However, it must be present at both ends of the connection,

so it is not practical for connections with external third parties. In a LAN environment, a similar solution using a “reverse NAT” box may be applicable in some cases, with the client host being oblivious to address changes, and the NAT middle box making sure that address changes do not affect applications. However, this too seems to require an infrastructure overhaul that we would prefer to avoid. All these practical constraints suggest that NASR should be implemented very carefully. A plausible scenario would involve NSR at the subnet level, and particularly for client hosts in DHCP-managed address pools.

2.2.2 Subnet address space utilization

The feasibility and effectiveness of NASR depend on the fraction of unused addresses in NASR-enabled subnets. Performing randomization on a sparse subnet will result in more connection failures for the hitlist worm compared to a dense subnet. Such failures could expose the worm as they could be picked up by scan-detection mechanisms. In a dense subnet with homogeneous systems (e.g., running the same services) the worm is more likely to succeed in infecting a host, even if the original host recorded in the hitlist has actually changed its address. Finally, in the extreme (and probably rare) case of a subnet that is always fully utilized, there will never be a free address slot to facilitate address changes. We attempt to get an estimate of typical subnet utilization levels. Because of the high scanning activity, we cannot perform this experiment globally without tripping a large number of alerts. We therefore opted for scanning five /16 prefixes that belong to FORTH, the University of Crete and a large ISP, after first obtaining permission by the administrators of the networks. We performed hourly scans on all prefixes using ICMP ECHO messages over a period of one month. A summary of the results is shown in Figure 3. For simplicity, we assume that all prefixes are subnetted in /24’s. We see that many subnets were completely dark with no hosts at all (not even a router). Nearly 30% of the subnets in two ISP prefixes were totally empty, while for the FORTH and UoC the percentage reaches 70%. This means that swapping subnets would likely be an effective NASR policy, but unfortunately it is not practical, as discussed in previous section. We also see that 95% of these subnets have less than 50% utilization and the number of maximum live hosts observed does not exceed 100. If subnet utilization at the global level is similar to what we see in our limited experiment, then NASR at the level of /24 subnets is likely to be quite effective, as there is sufficient room to move hosts around, reducing the effectiveness of the worm and causing it to make failed connections.

3. Emulation-based Polymorphic Attack Detection

The primary aim of an attacker or an Internet worm is to gain complete control over a target system. This is usually achieved by exploiting a vulnerability in a service running on the target system that allows the attacker to divert its flow of control and execute arbitrary code. The execution path of the vulnerable service can be diverted using several exploitation methods, such as buffer overflows, integer overflows, format string abuse, and arbitrary data corruption. The code that is executed after hijacking the instruction pointer is usually provided as part of the attack vector. Although the typical action of the injected code is to spawn a shell (hereby dubbed *shellcode*), the attacker can structure it to perform arbitrary actions under the privileges of the service that is being exploited [SK04]. For example, the "shellcode" of recent worms usually just connects back to the previous victim, downloads the main body of the worm, and executes it. In this work we use the term shellcode to refer to malicious injected code with any purpose.

Significant progress has been made in recent years towards detecting previously unknown code injection attacks at the network level [KIM04, SINGH04, NEWSOME05, TANG05, WANG04, KRUEGEL05, CHINCHANI05, WANG06, LI06]. However, as organizations start deploying state-of-the-art detection technology, attackers are likely to react by employing advanced evasion techniques, such as polymorphism and metamorphism, known from the virus scene since the early 1990s [SZOR05], to defeat these defenses.

Polymorphic shellcode engines create different forms of the same initial shellcode by encrypting its body with a different random key each time, and by prepending to it a decryption routine that makes it self-decrypting. Since the decryptor itself cannot be encrypted, some intrusion detection systems rely on the identification of the decryption routine of polymorphic shellcodes. While naive encryption engines produce constant decryptor code, advanced polymorphic engines mutate the decryptor using metamorphism [SZOR01], which collectively refers to techniques such as dead-code insertion, code transposition, register reassignment, and instruction substitution [CHRISTODORESCU03], making the decryption routine difficult to fingerprint.

A major outstanding question in security research and engineering is thus whether we can proactively develop mechanisms for the automatic containment of advanced polymorphic attacks. While results have been promising, and some approaches can cope with limited polymorphism, when polymorphism and metamorphism is combined with advanced evasion techniques like self-modifying code, as we demonstrate in the following, most of the existing proposals can be easily defeated.

3.1 Static Analysis Resistant Polymorphic Shellcode

Several research efforts have turned to static binary code analysis for detecting previously unknown polymorphic code injection attacks at the network level [KRUEGEL05, CHINCHANI05, WANG06, TOTH02, AKRITIDIS05, PAYER05]. These approaches treat the input network stream as potential machine code and analyze it for signs of mali-

cious behavior. The first step of the analysis involves the decoding of the binary machine instructions into their corresponding assembly language representation, a process called disassembly. Some methods rely solely to disassembly for identifying long instruction chains that may denote the existence of a NOP sled [TOTH02, AKRITIDIS05] or shellcode [PAYER05]. After the code has been disassembled, some techniques derive further control or data flow information that is then used for the discrimination between shellcode and benign data [KRUEGEL05, CHINCHANI05, WANG06].

However, after the flow of control reaches the shellcode, the attacker has complete freedom to structure it in a complex way that can thwart attempts to statically analyze it. In this section, we discuss ways in which polymorphic code can be obfuscated for evading network-level detection methods based on static binary code analysis.

Note that the techniques presented here are rather trivial, compared to elaborate binary code obfuscation methods [LINN03, AYCOCK05, VENABLE05], but powerful enough to illustrate the limitations of detection methods based on static analysis. Advanced techniques for complicating static analysis have also been extensively used for tamper-resistant software and for preventing the reverse engineering of executables, as a defense against software piracy [COLLBERG02, WANG00, MADOU05].

3.1.1 Thwarting Disassembly

There are two main disassembly techniques: *linear sweep* and *recursive traversal* [SCHWARZ02]. Linear sweep begins with the first byte of the stream and decodes each instruction sequentially, until it encounters an invalid opcode or reaches the end of the stream. The main advantage of linear sweep is its simplicity, which makes it very lightweight, and thus an attractive solution for high-speed network-level detectors.

Since the IA-32 instruction set is very dense, with 248 out of the 256 possible byte values representing a legitimate starting byte for an instruction, disassembling random data is likely to give long instruction sequences of seemingly legitimate code [PRASAD03]. The main drawback of linear sweep is that it cannot distinguish between code and data embedded in the instruction stream, and incorrectly interprets them as valid instructions [KRUEGEL04]. An attacker can exploit this weakness and evade detection methods based on linear sweep disassembly using well-known anti-disassembly techniques. The injected code can be obfuscated by interspersing junk data among the exploit code, not reachable at runtime, with the purpose to confuse the disassembler. Other common anti-disassembly techniques include overlapping instructions and jumping into the middle of instructions [COHEN93].

The recursive traversal algorithm overcomes some of the limitations of linear sweep by taking into account the control flow behavior of the program. Recursive traversal operates in a similar fashion to linear sweep, but whenever a control transfer instruction is encountered, it determines all the potential target addresses and proceeds with disassembly at those addresses recursively. For instance, in case of a conditional branch, it considers both the branch target and the instruction that immediately follows the jump. In this way,

it can “jump around” data embedded in the instruction stream which are never reached during execution.

```
0000  6A7F        push 0x7F
0002  59          pop ecx
0003  E8FFFFFFFF  call 0x7
0008  C15E304C    rcr [esi+0x30],0x4C
000C  0E          push cs
000D  07          pop es
000E  E2FA        loop 0xA
0010
...   <encrypted payload>
008F
```

(a)

```
0000  6A7F        push 0x7F
0002  59          pop ecx
0003  E8FFFFFFFF  call 0x7
0007  FFC1        inc ecx
0009  5E          pop esi
000A  304C0E07    xor [esi+ecx+0x7],cl
000E  E2FA        loop 0xA
0010
...   <encrypted payload>
008F
```

(b)

Figure 4: Disassembly of the decoder produced by the Countdown shellcode encryption engine using (a) linear sweep and (b) recursive traversal.

Figure 4 shows the disassembly of the decoder part of a shellcode encrypted using the Countdown encryption engine of the Metasploit Framework [METASPLOIT06] using linear sweep and recursive traversal. The code has been mapped to address `0x0000` for presentation purposes. The target of the `call` instruction at address `0x0003` lies at address `0x0007`, one byte before the end of `call`, i.e., the `call` instruction jumps to itself. This tricks linear disassembly to interpret the instructions immediately following the `call` instruction incorrectly. In contrast, recursive traversal follows the branch target and disassembles the overlapping instructions correctly.

However, the targets of control transfer instructions are not always identifiable. Indirect branch instructions transfer control to the address contained in a register operand and their destination cannot be statically determined. In such cases, recursive traversal also does not provide an accurate disassembly, and thus, an attacker could use indirect branches extensively to hinder it. Although some advanced static analysis methods can heuristically recover the targets of indirect branches, e.g., when used in jump tables, they are effective only with compiled code and well-structured binaries [SCHWARZ02, KRUEGEL04, CIFUENTES95, BALAKRISHNAN04]. A motivated attacker can construct highly obfuscated code that abuses any assumptions about the structure of the code, including the extensive use of indirect branch instructions, which impedes both disassembly methods.

3.1.2 Thwarting Control and Dataflow Analysis

Once the code has been disassembled, some approaches analyze the code further using *control flow analysis*, by extracting its control flow graph (CFG). The CFG consists of basic blocks as nodes, and potential control transfers between blocks as edges. Kruegel et al. [KRUEGEL05] use the CFG of several instances of a polymorphic worm to detect structural similarities between different mutations. Chinchani et al. [CHINCHANI05] differentiate between data and exploit code in network streams based on the control flow of the extracted code.

SigFree, proposed by Wang et al. [WANG06], uses both control and *data flow analysis* to discriminate between code and data. Data flow analysis examines the data operands of instructions and tracks the operations that are performed on them within a certain code block. After the extraction of the control flow graph, SigFree uses data flow analysis techniques to prune seemingly useless instructions, aiming to identify an increased number of remaining useful instructions that denote the presence of code.

However, even if a precise approximation of the CFG can be derived in the presence of indirect jumps or other anti-disassembly tricks, a motivated attacker can still hide the real CFG of the shellcode using *self-modifying* code, a much more powerful technique. Self-modifying code modifies its own instructions dynamically at runtime. Although payload encryption is also a form of self-modification, in this section we consider modifications to the decoder code itself, which is the only shellcode part exposed to static binary code analysis.

Since self-modifying code can transform almost any instruction of itself to a different instruction, an attacker can construct a decryptor that will eventually execute instructions that do not appear in the initial code image, on which static analysis methods operate on. Thus, crucial control transfer or data manipulation instructions can be concealed behind fake instructions, specifically selected to hinder control and data flow analysis. The real instructions will be written into the shellcode's memory image while it is executing, and thus are inaccessible to static binary code analysis methods.

```
0000  6A7F        push 0x7F
0002  59          pop ecx
0003  E8FFFFFFFF  call 0x7
0007  FFC1        inc ecx
0009  5E          pop esi
000a  80460AE0    add [esi+0xA],0xE0
000e  304C0E0B    xor [esi+ecx+0xB],cl
0012  02FA        add bh,dl
0014
...   <encrypted payload>
0093
```

Figure 5: A modified, static analysis resistant version of the Countdown decoder.

A very simple example of this technique, also known as "patching," is presented in Fig. 5, which shows the recursive traversal disassembly of a modified version of the Countdown decoder presented in Fig. 4. There are two main differences: an `add` instruction has been added at address `0x000A`, and the `loop 0xA` instruction has been replaced by `add bh,dl`. At first sight, this code does not look like a polymorphic decryptor, since the flow of control is linear, without any backward jumps that would form a decryption loop. However, in spite of the intuition we get by statically analyzing the code, the code is indeed a polymorphic decryptor which decrypts the encrypted payload correctly, as shown by the execution trace of Fig. 6.

```
0000  6A7F        push 0x7F
0002  59          pop ecx                 ;ecx = 0x7F
0003  E8FFFFFFFF  call 0x7                ;PUSH 0x8
0007  FFC1        inc ecx                 ;ecx = 0x80
0009  5E          pop esi                 ;esi = 0x8
000a  80460AE0    add [esi+0xA],0xE0      ;ADD [0012] 0xE0
000e  304C0E0B    xor [esi+ecx+0xB],cl    ;XOR [0093] 0x80
0012  E2FA        loop 0xE                ;ecx = 0x7F
000e  304C0E0B    xor [esi+ecx+0xB],cl    ;XOR [0092] 0x7F
0012  E2FA        loop 0xE                ;ecx = 0x7E
```

Figure 6: Execution trace of the modified Countdown decoder.

The decoder starts by initializing `ecx` with the value `0x7F`, which corresponds to the encoded payload size minus one. The `call` instruction sets the instruction pointer to the relative offset `-1`, i.e., the `inc ecx` instruction at address `0x0007`. `Pop` then loads the return address that was pushed in the stack by `call` in `ecx`. These instructions are used to find the absolute address from which the decoder is executing.

The crucial point is the execution of the `add [esi+0xA],0xE0` instruction. The effective address of the left operand corresponds to address `0x0012`, so `add` will modify its contents. Initially, at this address is stored the instruction `add bh,dl`. By adding the value `0xE0` to this memory location, the code at this location is modified and `add bh,dl` is transformed to `loop 0xe`. Thus, while the decryptor is executing, as soon as the instruction pointer reaches the address `0x0012`, the instruction that will actually be executed is `loop 0xe`.

Even in this simple form, the above technique is very effective in obfuscating the real CFG of the shellcode. Indeed, as shown in Fig. 7, a slight self-modification of just one instruction results to significant differences between the CFG derived using static analysis, and the actual CFG of the code that is eventually executed. If such self-modifications are applied extensively, then the real CFG can effectively be completely concealed. Going one step further, an attacker could implement a polymorphic engine that produces decryptors with arbitrarily fake CFGs, different in each shellcode instance, for evading detection methods based on CFG analysis. This can be easily achieved by placing fake control transfer instructions which during execution are overwritten with other useful instructions. Instructions that manipulate crucial data can also be concealed in the same manner in order to hinder data flow analysis. Static binary code analysis would need to be able to compute the output of each instruction in order to extract the real control and data flow of the code that will be eventually executed.

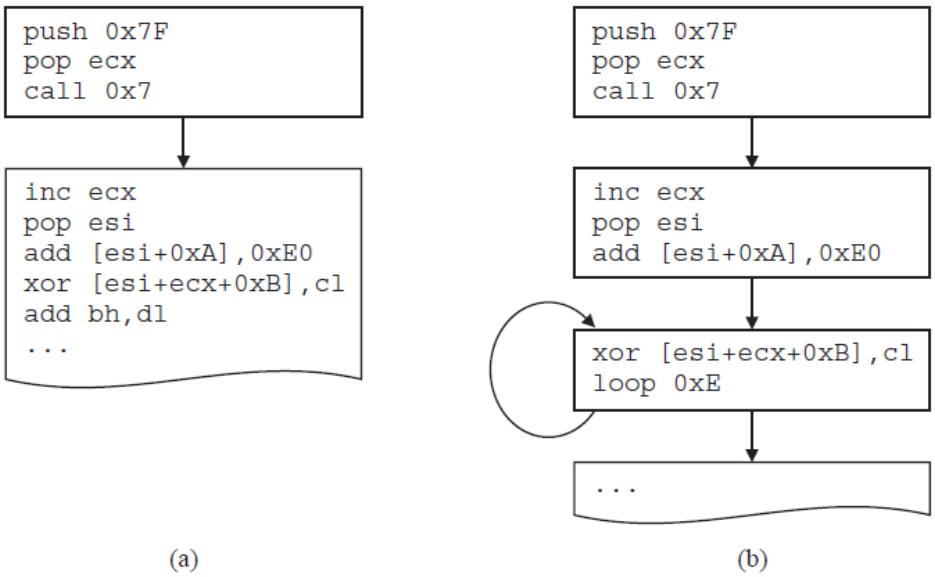

Figure 7: Control flow graph of the modified Countdown decoder (a) based on the code derived using recursive tra-versal disassembly, and (b) based on its actual execution.

3.2 Emulation-based Polymorphic Shellcode Detection

Carefully crafted polymorphic shellcode can evade detection methods based on static binary code analysis. Using anti-disassembly techniques, indirect control transfer instructions, and most importantly, self-modifications, static analysis resistant polymorphic shellcode will not reveal its actual form until it is eventually executed on a real CPU. This observation motivated us to explore whether it is possible to detect such highly obfuscated shellcode by actually executing it on a CPU emulator.

3.2.1 Approach

Our goal is to detect network streams that contain polymorphic exploit code by passively monitoring the incoming network traffic. Each request to some network service hosted in the protected network is treated as a potential attack vector. The detector attempts to execute each incoming request in a virtual environment as if it was executable code. Depending on the execution behavior, we can differentiate between benign data and polymorphic shellcode. Besides the NOP sled, which might not exist at all [23], the only executable part of polymorphic shellcodes is the decryption routine. Therefore, the detection algorithm focuses on the identification of the decryption process that takes place during the initial execution steps of a polymorphic shellcode.

In this work, we focus on the detection of polymorphic shellcodes. The execution of a polymorphic shellcode can be conceptually split in two sequential parts: the execution of the decryptor, and the execution of the actual payload. The accurate execution of the payload, which usually includes several advanced operations such as the creation of sockets or files, would require a complete virtual machine environment, including an appropriate

OS. In contrast, the decryptor is in essence a series of machine instructions that perform a certain computation over the memory locations where the encrypted shellcode has been injected. This allows us to simulate the execution of the decryptor using merely a CPU emulator. The only requirement is that the emulator should be compatible with the hardware architecture of the vulnerable host. For our prototype, we have focused on the IA-32 architecture.

The construction of polymorphic shellcodes conforms to several restrictions that allow us to simulate the execution of the decryptor part, even without having any further information about the context in which it is destined to run. In the remainder of this section we discuss these restrictions.

3.2.1.1 Position-independent code

In a dynamically changing stack or heap, the exact memory location where the shellcode will be placed is not known in advance. For this reason, any absolute addressing is avoided and reliable shellcode is made completely relocatable, in order to run from any memory position. Otherwise, the exploit becomes fragile [1]. For instance, in case of Linux stack-based buffer overflows, the absolute address of the vulnerable buffer varies between systems, even for the same compiled executable, due to the different environment variables that are stored in the beginning of the stack. This position-independent nature of shellcode allows us to map it in an arbitrary memory location and start its execution from there.

3.2.1.2 GetPC code

Both the decryptor and the encrypted payload are part of the injected vector, with the decryptor stub usually prepended to the encrypted payload. Since the absolute memory address of the injected shellcode cannot be accurately predicted in advance, the decoder needs to somehow find a reference to this exact memory location in order to decrypt the encrypted payload.

To this end, shellcodes take advantage of the CPU program counter (PC, or EIP in the IA-32 architecture). During the execution of the decryptor, the PC points to the decryptor code, i.e., to an address within the memory region where the decryptor, along with the encrypted payload, has been placed. However, the IA-32 architecture does not provide any EIP-relative memory addressing mode,[1] as opposed to instruction dispatch. Thus, the decryptor cannot use the PC directly to reference to the memory locations of the encrypted payload in order to modify it. Instead, the decryptor first loads the current value of the PC to a register, and then uses this value to compute the absolute address of the payload. The code that is used for retrieving the current PC value is usually referred to as the “getPC” code.

The simplest way to read the value of the PC is through the use of the `call` instruction. The intended use of `call` is for calling a procedure. When the `call` instruction is exe-

[1] The IA-64 architecture supports a RIP-relative data addressing mode. RIP stands for the 64bit instruction

cuted, the CPU pushes the return address in the stack, and jumps to the first instruction of the called procedure. The return address is the address of the instruction immediately following the `call` instruction. Thus, the decryptor can compute the address of the encrypted payload by reading the return address from the stack and adding to it the appropriate offset in order to reference the payload memory locations. This technique is used by the decryptor shown in Fig. 4. The encrypted payload begins at address `0x0010`. `Call` pushes in the stack the address of the instruction immediately following it (`0x0008`), which is then popped to `esi`. The size of the encrypted payload is computed in `ecx`, and the effective address computation `[esi+ecx+0x7]` in `xor` corresponds to the last byte of the encrypted payload at address `0x08F`. As the name of the engine implies, the decryption is performed backwards, one byte at a time, starting from the last encrypted byte.

```
0000  6A04            push byte +0x4
0002  59              pop ecx
0003  D9EE            fldz
0005  D97424F4        fnstenv [esp-0xc]
0009  5B              pop ebx
000A  817313CACD4B2E  xor dword [ebx+0x13],0x2e4bcdca
0011  83EBFC          sub ebx,byte -0x4
0014  E2F4            loop 0xa
...
```

Figure 8: The decryptor of the PexFnstenvMov engine, which is based on a getPC code that uses the fnstenv instruction.

Finding the absolute memory address of the decryptor is also possible using the `fstenv` instruction, which saves the current FPU operating environment at the memory location specified by its operand [NOIR03]. The stored record includes the instruction pointer of the FPU, which is different than EIP. However, if a floating point instruction has been executed as part of the decryptor, then the FPU instruction pointer will also point to the memory area of the decryptor, and thus `fstenv` can be used to retrieve its absolute memory address. The same can also be achieved using one of the `fstenv`, `fsave`, or `fnsave` instructions.

Figure 8 shows the decoder generated by the PexFnstenvMov engine of the Metasploit Framework [METASPLOIT06], which uses an `fnstenv`-based getPC code. By specifying the memory offset to the `fstenv` relative to the stack pointer, the absolute memory address of the latest floating point instruction `fldz` can be popped to `ebx`. By combining the `fstenv`-based getPC code with self-modifications, it is possible to construct a decoder with no control transfer instruction, i.e., with a CFG consisting of a single node.

A third getPC technique is possible by exploiting the structured exception handling (SEH) mechanism of Windows [IONESCU03]. However this technique works only with older versions of Windows, and the introduction of registered SEH in Windows XP and 2003 limits its applicability. From the tested polymorphic shellcode engines, only Alpha2 [WEVER04] supports this type of getPC, although not by default.

3.2.1.3 Known operand values

Polymorphic shellcode engines produce generic decryptor code for a specific hardware platform that runs independently of the OS version of the victim host or the vulnerability being exploited. The decoder is constructed with no assumptions about the state of the process in which it will run, and any registers or memory locations being used by the decoder are initialized on the fly. This allows us to correctly follow its execution from the very first instruction, since instruction operands with initially unknown values will eventually become available.

For instance, the execution trace of the Countdown decoder in Fig. 6 is always the same, independently of the process in which it has been injected. Indeed, the code is *self-contained*, which allows us to correctly execute even instructions with non-immediate operands which otherwise would be unknown, as shown from the comments next to the code. The emulator can correctly initialize the registers, follow stack operations, compute all effective addresses, and even follow self modifications, since every operand eventually becomes known.

3.2.2 Detection Algorithm

In this section we describe in detail the emulation-based polymorphic shellcode detection algorithm. The algorithm takes as input a byte stream captured passively from the network, such as a reassembled TCP stream or the payload of a UDP packet, and reasons whether it contains polymorphic shellcode. Each input is executed on a CPU emulator as if it was executable code. Due to the dense instruction set and the variable instruction length of the IA-32 architecture, even non-attack streams can be interpreted as valid executable code. However, such random code usually stops running soon, e.g., due to the execution of an illegal instruction, while real polymorphic code is being executed until the encrypted payload is fully decrypted.

The pseudo-code of the detection algorithm is presented in Fig. 9 with several simplifications for brevity. Each input buffer is mapped to a random location in the virtual address space of the emulator, as shown in Fig. 10. This corresponds to the placement of the attack vector into the input buffer of a vulnerable process. Before each execution attempt, the state of the virtual processor is randomized (line 5). Specifically, the `EFLAGS` register, which holds the flags of conditional instructions, and all general purpose registers are assigned random values, except `esp`, which is set to point to the middle of the stack of a supposed process.

3.2.2.1 Running the shellcode

The main routine, `emulate`, takes as parameters the starting address and the length of the input stream. Depending on the vulnerability, the injected code may be located at an arbitrary position within the stream. For example, the first bytes of a TCP stream or a UDP packet payload will probably be occupied by protocol data, depending on the application (e.g., the `METHOD` field in case of an HTTP request). Since the position of the

shellcode is not known in advance, the main routine consists of a loop which repeatedly starts the execution of the supposed code that begins from each and every position of the input buffer (line 3). We call a complete execution starting from position *i* an *execution chain from i.*

```
1    emulate(buf_start_addr, buf_len) {
2        invalidate_translation_cache();
3        for (pos=buf_start_addr; pos<buf_len; ++pos) {
4            PC = pos;
5            reset_CPU();
6            do {
7                /* decode instruction if no entry in translation cache */
8                if (translation_cache[PC] == NULL)
9                    translation_cache[PC] = decode_instruction(buf[PC]);
10               if (translation_cache[PC] == (ILLEGAL || PRIVILEGED)
11                   break;
12               execute(translation_cache[PC]); /* changes PC */
13               if (vmem[PC] == INVALID)
14                   break;
15           }
16           while (num_exec++ < XT);
17           if (has_getPC_code && (payload_reads >= PRT)
18               return TRUE;
19       }
20       return FALSE;
21   }
```

Figure 9: Simplified pseudo-code for the detection algorithm.

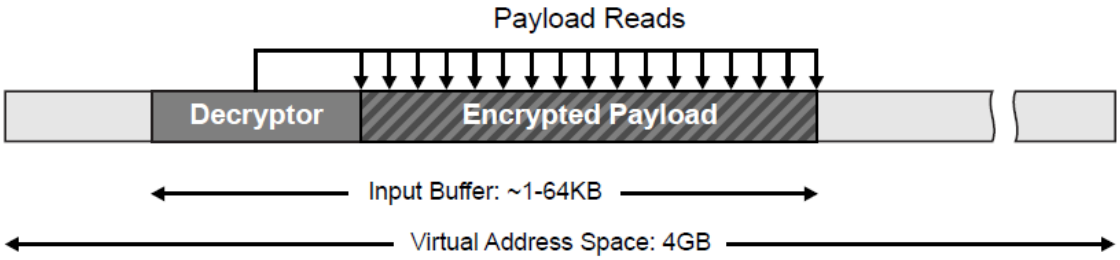

Figure 10: Memory reads during the decryption of a polymorphic shellcode.

Note that it is necessary to start the execution from each position *i*, instead of starting only from the first byte of the stream and relying on the self-synchronizing property of the IA-32 architecture [KRUEGEL05, CHINCHANI05], since we may otherwise miss the execution of a crucial instruction that initializes some register or memory location. For example, going back to the execution trace of Fig. 6, if the execution misses the first instruction `push 0xF`, e.g., due to a misalignment or an overlapping instruction placed in purpose immediately before `push`, then the emulator will not execute the decryptor correctly, since the value of the `ecx` register will be arbitrary. Furthermore, the execution may stop even before reaching the shellcode, e.g., due to an illegal instruction.

For each position `pos`, the algorithm enters the main execution loop (line 6), in which a new instruction is fetched, decoded, the program counter is increased by the length of the

instruction, and finally the instruction is executed. In case of a control transfer instruction, upon its execution, the PC might have been changed to the address of the target instruction. Since instruction decoding is an expensive operation, decoded instructions are stored in a translation cache (line 9). If an instruction at a certain position of the buffer is going to be executed again, e.g., as part of a loop body in the same execution chain, or as part of a different execution chain of the same input buffer then the instruction is instantly fetched from the translation cache.

3.2.2.2 Detection heuristic

While the execution behavior of random code is undefined, there exists a generic execution pattern inherent to all polymorphic shellcodes, which allows us to accurately distinguish polymorphic code injection attacks from benign requests. Upon the hijack of the program counter, the control flow of the vulnerable process is diverted—sometimes through a NOP sled—to the injected shellcode, and particular to the polymorphic decryptor. During decryption, the decryptor reads the contents of the memory locations where the encrypted payload has been stored, decrypts them, and writes back the decrypted data. Hence, the decryption process will result in many memory accesses to the memory region where the input buffer has been mapped to. Since this region is a very small part of the virtual address space, we expect that memory reads from that area would occur rarely during the execution of random code.

Only instructions with a memory operand can potentially result in a memory read from the input buffer. This may happen if the absolute address that is specified by a direct memory operand, or if the computation of the effective address of an indirect memory operand, corresponds to an address within the input buffer. Input streams are mapped to a random memory location of the 4GB virtual address space. Additionally, before each execution, the CPU registers, some of which normally take part in the computation of the effective address, are randomized. Thus, the probability to encounter an accidental read from the memory area of the input buffer in random code is very low. In contrast, the decryptor will access tens or hundreds of *different* memory locations within the input buffer, as depicted in Fig. 10, depending on the size of the encrypted payload and the decryption function.

This observation led us to initially choose the number of reads from *distinct* memory locations of the input buffer as the detection criterion. For the sake of brevity, we refer to memory reads from distinct locations of the input buffer as *"payload reads."* For a given execution chain, a number of payload reads greater than a certain payload reads threshold (PRT) gives an indication for the execution of a polymorphic shellcode. We expected random code to exhibit a low payload reads frequency, which would allow for a small PRT value, much lower than the typical number of payload reads found in polymorphic shellcodes. Preliminary experiments with network traces showed that the frequency of payload reads in random code is very small, and usually only a few of the incoming streams had execution chains with just one to ten payload reads. However, there were rare cases with execution chains that performed hundreds of payload reads. This was usually due to the accidental formation of a loop with an instruction that happened to read

hundreds of different memory locations from the input buffer. Since we expected random code to exhibit a low number of payload reads, such behavior would have been flagged as polymorphic shellcode by our initial criterion, which would result in false positives.

Since one of our primary goals is to have practically zero false positives, we addressed this issue by defining a more strict criterion. A mandatory operation of every polymorphic shellcode is to find its absolute memory address through the execution of some form of getPC code. This led us to augment the detection criterion as follows: if an execution chain of an input stream executes some form of getPC code, followed by PRT or more payload reads, then the stream is flagged to contain polymorphic shellcode.

Another option for enhancing the detection heuristic would be to look for *linear* payload reads from a contiguous region of the input buffer. However, this heuristic can be tricked by splitting the encrypted payload into nonadjacent parts which can then be decrypted in random order [PERRIOT02].

3.2.2.3 Ending execution

An execution chain may end for one of the following reasons: (i) an illegal or privileged instruction is encountered, (ii) the control is transferred to an invalid or unknown memory location, or (iii) the number of executed instructions has exceeded a certain threshold.

- **Invalid instruction.** The execution may stop if an illegal or privileged instruction is encountered (line 10). Since privileged instructions can be invoked only by the OS kernel, they cannot take part in the normal shellcode execution. Although an attacker could intersperse invalid or privileged instructions in the injected code to hinder detection, these should come with corresponding control transfer instructions that would bypass them during execution—otherwise the shellcode would not execute correctly. In that case, the emulator will also follow the real execution of the code, so such instructions will not cause any inconsistency. At the same time, privileged or illegal instructions appear relatively often in random data, helping this way the detector to distinguish between benign requests and attack vectors.

- **Invalid memory location.** Normally, during the execution of the decoder, the program counter will point to addresses of the memory region of the input buffer where the injected code resides. However, highly obfuscated code could use the stack for storing some parts, or all of the decrypted code, or even for "producing" useful instructions on the fly, in a way similar to the self-modifications presented in previous sections. Thus, the flow of control may jump from the original code of the decryptor to some generated instruction in the stack, then jump back to the input buffer, and so on. In fact, since the shellcode is the last piece of code that will be executed as part of the vulnerable process, the attacker has the flexibility to write in *any* memory location mapped in the address space of the vulnerable process [OBSCOU03]. Although it is generally difficult to know in advance the contents of a certain memory location, since they usually vary between different sys-

tems, it is easier to find virtual memory regions that are always mapped into the address space of the vulnerable process. For example, if address space randomization is not applied extensively, the attacker might know in advance some memory regions of the stack or heap that exist in every instance of the vulnerable process.

- **Execution threshold.** There are situations in which the execution of random code might not stop soon, or even not at all, due to large code blocks with no backward branches that are executed linearly, or due to the occurrence of backwards jumps that form seemingly "endless" or infinite loops. In such cases, an execution threshold (XT) is necessary for avoiding extensive performance degradation or execution hang ups (line 16). An attacker could exploit this and evade detection by placing a loop before the decryptor which would execute enough instructions to exceed the execution threshold before the code of the actual decryptor is reached. We cannot simply skip such loops, since the loop body could perform a crucial computation for the further correct execution of the decoder, e.g., computing the decryption key. Fortunately, endless loops occur with low frequency in normal traffic. Thus, an increase in input requests with execution chains that reach the execution threshold due to a loop might be an indication of a new attack outbreak using the above evasion method.

4. Conclusion

In this document, we have presented the design of the two main defense approaches being developed in the research project MILTIADES.

IP address space randomization hinders the propagation of hitlist worms by making the contents of the precomputed hit list stale, thereby leading the worm to make several unsuccessful attempts to compromise non-vulnerable computers. These unsuccessful connections will probably be more visible to firewalls and intrusion detection systems, which will quickly take notice of the spreading worm.

Emulation-based polymorphic attack detection is a novel detection approach against previously unknown polymorphic attacks, which is based on the actual execution of attack data on a CPU emulator. The actual execution of the attack code on a CPU emulator makes the detector robust to evasion techniques such as highly obfuscated or self-modifying code, while each shellcode is detected separately, which makes it effective against targeted attacks. Furthermore, the detector does not rely on any exploit or vulnerability specific signatures, which allows the detection of previously unknown attacks.

Bibliography

- [SK04] sk, “History and advances in windows shellcode,” Phrack, vol. 11, no. 62, July 2004.
- [KIM04] H.-A. Kim and B. Karp, “Autograph: Toward automated, distributed worm signature detection,” in Proceedings of the 13th USENIX Security Symposium, 2004, pp. 271–286.
- [SINGH04] S. Singh, C. Estan, G. Varghese, and S. Savage, “Automated worm fingerprinting,” in Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI), Dec. 2004.
- [NEWSOME05] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatically generating signatures for polymorphic worms,” in Proceedings of the IEEE Security & Privacy Symposium, May 2005, pp. 226–241.
- [TANG05] Y. Tang and S. Chen, “Defending against internet worms: a signature-based approach,” in Proceedings of the 24th Annual Joint Conference of IEEE Computer and Communication societies (INFOCOM), 2005.
- [WANG04] K.Wang and S. J. Stolfo, “Anomalous payload-based network intrusion detection,” in Proceedings of the 7th International Symposium on Recent Advanced in Intrusion Detection (RAID), September 2004, pp. 201–222.
- [KRUEGEL05] C. Kruegel, E. Kirda, D. Mutz,W. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), Sept. 2005.
- [CHINCHANI05] R. Chinchani and E. V. D. Berg, “A fast static analysis approach to detect exploit code inside network flows,” in Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), Sept. 2005.
- [WANG06] X. Wang, C.-C. Pan, P. Liu, and S. Zhu, “Sigfree: A signature-free buffer overflow attack blocker,” in Proceedings of the USENIX Security Symposium, Aug. 2006.
- [LI06] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, “Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience,” in Proceedings of the 2006 IEEE Symposium on Security and Privacy, 2006, pp. 32–47.
- [SZOR05] P. Szor, The Art of Computer Virus Research and Defense. Addison-Wesley Professional, February 2005.
- [SZOR01] P. Szor and P. Ferrie, “Hunting for metamorphic,” in Proceedings of the Virus Bulletin Conference, Sept. 2001, pp. 123–144.
- [CHRISTODORESCU03] M. Christodorescu and S. Jha, “Static analysis of executables to detect malicious patterns,” in Proceedings of the 12th USENIX Security Symposium (Security’03), Aug. 2003.
- [TOTH02] T. Toth and C. Kruegel, “Accurate buffer overflow detection via abstract payload execution,” in Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID), Oct. 2002.

- [AKRITIDIS05] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis, "STRIDE: Polymorphic sled detection through instruction sequence analysis," in Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC), June 2005.
- [PAYER05] U. Payer, P. Teufl, and M. Lamberger, "Hybrid engine for polymorphic shellcode detection," in Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2005, pp. 19–31.
- [LINN03] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in Proceedings of the 10th ACM conference on Computer and communications security (CCS), 2003, pp. 290–299.
- [AYCOCK05] J. Aycock, R. deGraaf, and M. Jacobson, "Anti-disassembly using cryptographic hash functions," Department of Computer Science, University of Calgary, Tech. Rep. 2005-793-24.
- [VENABLE05] M. Venable, M. R. Chouchane, M. E. Karim, and A. Lakhotia, "Analyzing memory accesses in obfuscated x86 executables," in Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2005.
- [COLLBERG02] C. S. Collberg and C. Thomborson, "Watermarking, tamperproffing, and obfuscation: tools for software protection," IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 735–746, 2002.
- [WANG00] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," University of Virginia, Tech. Rep. CS-2000-12, 2000.
- [MADOU05] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. D. Sutter, and K. D. Bosschere, "Software protection through dynamic code mutation," in Proceedings of the 6th International Workshop on Information Security Applications (WISA), Aug. 2005, pp. 194–206.
- [SCHWARZ02] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of executable code revisited," in Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE), 2002.
- [PRASAD03] M. Prasad and T. cker Chiueh, "A binary rewriting defense against stack based overflow attacks," in Proceedings of the USENIX Annual Technical Conference, June 2003.
- [KRUEGEL04] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in Proceedings of the USENIX Security Symposium, Aug. 2004, pp. 255–270.
- [COHEN93] F. B. Cohen, "Operating system protection through program evolution," Computer and Security, vol. 12, no. 6, pp. 565–584, 1993.
- [METASPLOIT06] "Metasploit project," 2006, http://www.metasploit.com/.
- [CIFUENTES95] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," Software—Practice and Experience, vol. 25, no. 7, pp. 811–829, 1995.
- [BALAKRISHNAN04] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in Proceedings of the International Conference on Compiler Construction (CC), Apr. 2004.

- [NOIR03] Noir, “GetPC code (was: Shellcode from ASCII),” June 2003, http://www.securityfocus.com/archive/82/327100/2006-01-03/1.
- [IONESCU03] C. Ionescu, “GetPC code (was: Shellcode from ASCII),” July 2003, http://www.securityfocus.com/archive/82/327348/2006-01-03/1.
- [WEVER04] B.-J. Wever, “Alpha 2,” 2004, http://www.edup.tudelft.nl/»bjwever/src/alpha2.c.
- [PERRIOT02] F. Perriot, P. Ferrie, and P. Sz¨or, “Striking similarities,” Virus Bulletin, pp. 4–6, May 2002.
- [OBSCOU03] Obscou, “Building IA32 ‘unicode-proof’ shellcodes,” Phrack, vol. 11, no. 61, Aug. 2003.
- [CODERED01] CERT Advisory CA-2001-19: ‘Code Red’ Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. http://www.cert.org/advisories/CA-2001-19.html, July 2001.
- [SLAMMER03] “The Spread of the Sapphire/Slammer Worm”. http://www.silicondefense.com/research/worms/slammer.php, February 2003.
- [ATIG03] Michael Atighetchi, Partha Pal, Franklin Webber, Rick Schantz, and Chris Jones. “Adaptive use of network-centric mechanisms in cyber-defense”. In Proceedings of the 6th IEEE International Symposium on Object-oriented Real-time Distributed Computing, May 2003.
- [BARATTO04] Ricardo A. Baratto, Shaya Potter, Gong Su, and Jason Nieh. “Mobidesk: mobile virtual desktop computing”. In Proceedings of the 10th Annual International Conference on Mobile Computing and Networking (MOBICOM), pages 1–15. ACM Press, 2004.
- [GAURAV03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. “Countering Code-Injection Attacks With Instruction-Set Randomization” . In Proceedings of the ACM Computer and Communications Security Conference (CCS), pages 272–280, October 2003.
- [MOORE02] D. Moore, C. Shannon, and J. Brown. “Code-Red: a case study on the spread and victims of an Internet worm”. In Proceedings of the 2nd Internet Measurement Workshop (IMW), pages 273–284, November 2002.
- [STANIFORD04] S. Staniford, D. Moore, V. Paxson, and N. Weaver. “The top speed of flash worms”. In Proc. ACM CCS WORM, October 2004.
- [STANIFORD02] S. Staniford, V. Paxson, and N. Weaver. “How to Own the Internet in Your Spare Time”. In Proceedings of the 11th USENIX Security Symposium, pages 149–167, August 2002.
- [WEAVER04] N. Weaver and V. Paxson. “A worst-case worm”. In Proc. Third Annual Workshop on Economics and Information Security (WEIS’04), May 2004.
- [WILLIAMSON02] M. Williamson. “Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code”. Technical Report HPL-2002-172, HP Laboratories Bristol, 2002