Network-Level Polymorphic Shellcode Detection Using Emulation

Michalis Polychronakis[1], Kostas G. Anagnostakis[2], and Evangelos P. Markatos[1]

[1] Institute of Computer Science, Foundation for Research & Technology – Hellas
{mikepo,markatos}@ics.forth.gr
[2] Internet Security Lab, Institute for Infocomm Research, Singapore
kostas@i2r.a-star.edu.sg

Technical Report 368
ICS-FORTH, January 2006

Abstract. As state-of-the-art attack detection technology becomes more prevalent, attackers are likely to evolve, employing techniques such as polymorphism and metamorphism to evade detection. Although recent results have been promising, most existing proposals can be defeated using only minor enhancements to the attack vector. We present a heuristic detection method that scans network traffic streams for the presence of polymorphic shellcode. Our approach relies on a NIDS-embedded CPU emulator that executes every potential instruction sequence, aiming to identify the execution behavior of polymorphic shellcodes. Our analysis demonstrates that the proposed approach is more robust to obfuscation techniques like self-modifications compared to previous proposals, but also highlights advanced evasion techniques that need to be more closely examined towards a satisfactory solution to the polymorphic shellcode detection problem.

1 Introduction

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 code that is executed after hijacking of 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* in the security community), the attacker can structure the injected code to perform arbitrary actions under the privileges of the service that has been exploited [1].

Significant progress has been made in recent years towards detecting previously unknown code injection attacks at the network level [2–8]. 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 community since the early 90s. 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 which makes it self-decrypting. Since the decryptor itself cannot be encrypted, some detection approaches rely on the identification of the decryption routine. Although simple encryption engines produce constant decryptor code, advanced polymorphic engines mutate the decryptor using metamorphism [9], which collectively refers to techniques such as dead-code insertion, code transposition, register reassignment, and instruction substitution [10], making the decryption routine difficult to fingerprint.

A major outstanding question in security research and engineering is thus whether we can proactively develop the tools needed to contain advanced polymorphic attacks. While results have been promising, most of the existing proposals can be easily defeated. In fact, publicly-available polymorphic engines [11] are currently one step ahead of the most advanced publicly-documented detection engines.

In this paper, we revisit the question of whether polymorphic shellcode is detectable at the network-level. We present a detection heuristic that tests whether byte sequences in network traffic have properties similar to polymorphism. Specifically, we speculatively execute potential instruction sequences and compare their execution profile against behavior observed to be inherent to polymorphic shellcodes. Our approach relies on a fully-blown IA-32 CPU emulator, which, in contrast to previous work, makes the detector immune to runtime evasion techniques such as self-modifying code.

The remainder of this report is organized as follows. Section 2 summarizes related work. Section 3 discusses techniques that can be used for evading detection methods based on static binary code analysis. We present in detail our detection approach in Section 4, and our experimental and performance results in Section 5. Finally, Section 6 discusses limitations of our approach, and Section 7 concludes the paper.

2 Related Work

Network intrusion detection systems (NIDS) like Snort [12] and Bro [13] have been extensively used to detect shellcodes, including previously unseen ones, using signatures that match components common to similar exploits, such as the NOP sled, protocol framing, or specific parts of the shellcode [14]. As a response, attackers started to employing polymorphism [11, 15, 16] for evading signature-based NIDS.

Initial approaches on zero-day polymorphic shellcode detection focused on the identification of the sled component [17, 18]. However, sleds are mostly useful in expediting exploit development, and in several cases, especially in Windows exploits, can be completely avoided through careful engineering using register springs [19]. In fact, most worms so far do not have sleds. Our approach focuses on the detection of the polymorphic shellcode itself and works even in the absence of the sled component. Buttercup [20] attempts to detect polymorphic buffer overflow attacks by identifying the ranges of the possible return addresses for existing buffer overflow vulnerabilities.

Several research efforts have focused on the automated generation of signatures for previously unknown worms based on the prevalence of common byte sequences across different worm instances, either by correlating payloads from different traffic flows [2, 3], or using honeypots [21]. However, these approaches are ineffective against polymorphic and metamorphic worms [9, 22]. Polygraph [4], PAYL [6], and PADS [5] attempt to detect polymorphic worms by identifying common invariants among different worm instances, such as return addresses, protocol framing, and poor obfuscation, and derive regular expression or statistical signatures. Although above approaches can identify simple obfuscated worms, their effectiveness is still questionable in the presence of extensive polymorphism [11]. Moreover, they require multiple worm instances before reasoning for a threat, which makes them ineffective against targeted attacks.

Having identified the limitations of signature-based approaches, recent research efforts, most closely related to our work, have turned to static binary code analysis for identifying exploit code in network flows. Payer et al. [23] describe a hybrid polymorphic shellcode detection engine based on a neural network that combines several heuristics, including a NOP-sled detector and recursive traversal disassembly. However, the neural network must be trained with both positive and negative data in order to achieve a good detection rate, which makes it ineffective against zero-day attacks. Kruegel et al. [7] present a worm detection method that identifies structural similarities between different worm mutations. In contrast, our approach can detect targeted polymorphic code-injection attacks from the first attack instance. Styx [8] differentiates between benign data and program-like exploit code in network streams by looking for meaningful data and control flow, and blocks identified attacks using automatically generated signatures. A fundamental limitation of such approaches based on static analysis of binary code is that an attacker can evade them using obfuscations such as self-modifying code, as we discuss in the following Section.

3 Static Analysis Resistant Polymorphic Shellcode

Several research efforts have been based on static binary code analysis for the detection of zero-day polymorphic shellcode injection attacks at the network level [7,8,17,18,23]. These approaches treat the input network stream as potential machine code which is then analyzed for detecting patterns of malicious behavior. Some methods rely solely to disassembly for identifying long instruction chains that may denote the existence of a NOP sled [17, 18] or shellcode [23], while others [7, 8] derive further control flow information, which is used for the discrimination between shellcode and benign data. However, after the flow of control reaches the shellcode, the attacker has complete freedom to structure it in a complex way that will thwart attempts to statically analyze it. In this section we discuss ways in which polymorphic code can be obfuscated for evading detection from network-level static binary code analysis methods.

Note that the techniques presented here are rather trivial, compared to elaborate obfuscation methods [24–26], but enough to illustrate the limitations of static analysis based detection methods. Advanced techniques for complicating static binary code analysis have also been extensively used for tamper-resistant software and for preventing the reverse engineering of executables, as a defense against software piracy [27–29].

3.1 Thwarting Disassembly

There are two main disassembly techniques: *linear sweep* and *recursive traversal* [30]. 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. Since the IA-32 instruction set is very dense, disassembling random data is

```
0000  6A0F          push 0x7F                0000  6A0F         push 0x7F
0002  59            pop ecx                  0002  59           pop ecx
0003  E8FFFFFFFF    call 0x7                 0003  E8FFFFFFFF   call 0x7
0008  C15E304C      rcr [esi+0x30],0x4C      0007  FFC1         inc ecx
000C  0E            push cs                  0009  5E           pop esi
000D  07            pop es                   000A  304C0E07     xor [esi+ecx+0x7],cl
000E  E2FA          loop 0xA                 000E  E2FA         loop 0xA
0010                                         0010
...   <encrypted payload>                    ...   <encrypted payload>
008F                                         008F
              (a)                                          (b)
```

Fig. 1. Disassembly of the decoder produced by the Countdown shellcode encryption engine using (a) linear sweep and (b) recursive traversal.

likely to give long instruction sequences of seemingly legitimate code [31]. 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 [32]. There are several well-known anti-disassembly techniques against linear sweep, such as interspersing junk data among the shellcode, not reachable at runtime, creating overlapping instructions, and jumping into the middle of instructions [33]. 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 but whenever a control transfer instruction is encountered, it determines all the potential target addresses and proceeds with disassembly at those addresses recursively.

Figure 1 shows the disassembly of the decoder part of a shellcode encrypted using the Countdown encryption engine of the Metasploit Framework [34] using linear sweep and recursive traversal. The target of the `call` instruction at address `0x0003` is address `0x0007`, one byte before the end of the `call` instruction, 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 correctly the overlapping instructions.

However, the targets of control transfer instructions are not always identifiable. Some indirect branch instructions transfer control to the address contained in a register operand, thus their destination cannot be 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. Some advanced static analysis methods can heuristically recover the targets of indirect branches with high accuracy, although they are usually effective only with well-structured binaries [30, 32, 35, 36].

3.2 Thwarting Control Flow Graph Extraction

Once the code has been disassembled, the next step of some detection methods is to perform analysis based on the control-flow of the code by extracting the Control Flow Graph (CFG). The CFG consists of basic blocks as nodes, and potential control transfers between blocks as edges. Control flow analysis provides a more distinctive representation of the program structure by considering essential information on control transfer. Kruegel et al. [7] use the CFG of several instances of a polymorphic worm to detect structural similarities between different mutations. Chinchani et al. [8] differen-

```
0000  6A0F          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
```

Fig. 2. Recursive traversal disassembly of the modified version of the Countdown decoder.

```
0000  6A0F          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
000e  304C0E0B      xor [esi+ecx+0xB],cl     ;XOR [0092] 0x79
0012  E2FA          loop 0xE
...
```

Fig. 3. Execution trace of the modified Countdown decoder.

tiate between data and exploit code in network streams based on the control flow of the extracted code.

Even if a precise approximation of the CFG can be derived in the presence of indirect jumps, e.g., by first identifying all branch instructions in the network stream [8], it is still possible for a motivated attacker to hide the real CFG of the shellcode using self-modifying code, which changes itself dynamically at runtime. Although the encryption of the payload 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 techniques. Using self-modifications, an attacker can construct a decryptor that will evenutally execute instructions that do not appear in the code derived using any static analysis method, but will be written into the shellcode's memory image after its execution has been started.

A very simple example of this technique, also known as "patching," is presented in Figure 2, which shows the recursive traversal disassembly of a modified version of the decoder presented in Figure 1. There are two main differences: an `add` instruction has been added at address `0x000A`, and `loop` has been replaced by the `add bh,dl` instruction. At first sight, this code does not look like a polymorphic decryptor, since the flow of control is linear, with no backward jumps that would form a decryption loop. However, the code decrypts the ecrypted payload correctly, as shown by the execution trace of Figure 3. The `add [esi+0xA],0xE0` instruction modifies the contents of address `0x0012`, which initially contains the instruction `add bh,dl`. By adding to this memory location the value `0xE0`, the code at this location is modified and `add bh,dl` is transformed to `loop 0xe`. Thus, when the instruction pointer reaches address `0x0012`, the instruction that is actually executed is `loop 0xe`.

Even in this simple form, the above technique is very effective in obfuscating the real CFG of shellcodes. Going one step further, an attacker could implement a polymorphic engine that produces decryptors with arbitrary fake CFGs, different in each shellcode instance, for evading detection methods based on CFG extraction. This can be easily achieved by placing fake control transfer instructions, which during execution are overwritten with other useful instructions. Static binary code analysis would need to be able to compute the output of each instruction in order to extract the real CFG of the code that will be eventually executed.

4 Network-level Execution

Carefully crafted polymorphic shellcode can evade detection methods based on static binary code analysis. Using anti-disassembly techniques, indirect control transfer instructions, and 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, using only information available at the network level.

4.1 Approach

Our goal is to detect network streams that contain polymorphic exploit code by passively monitoring the incoming network traffic. Requests to services hosted in the protected network are treated as potential attack vectors. The detector attempts to “execute” each incoming request in a virtual environment as if it was executable code. Besides the NOP sled, 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.

Being isolated from the vulnerable host, the detector lacks the context in which the injected code would run. Crucial information such as the OS of the host and which process is being exploited might not be known in advance. The execution of a polymorphic shellcode can be conceptually split into the execution of two sequential parts: the decryptor and 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 simply performs a certain computation over the memory locations of the encrypted payload. 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.

Though, the context of the vulnerable process in which the shellcode would be injected is still missing. Specifically, since the emulator has no access to the victim host, it lacks the memory and CPU state of the vulnerable process at the time its flow of control is diverted to the injected code. However, the construction of polymorphic shellcodes conforms to several restrictions, which allows us to simulate the execution of the decryptor part even with no further information about the context in which it is destined to run. In the remainder of this section we discuss these characteristics.

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, otherwise the exploit becomes fragile [1]. For example, 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 environment variables which are stored in the beginning of the stack area. Any shellcode exploiting such a vulnerability using some hardcoded stack address would become nonfunctional in most cases. The position-independent nature of shellcode allows us to map it in an arbitrary memory location and start its execution from there.

GetPC code. Both the decoder and the encoded payload are part of the injected vector, with the decoder 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 find some reference to that memory location in order to decrypt the encrypted payload. During its execution, the instruction pointer (EIP) points to the decryptor code, i.e., to some 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[3] (as opposed to instruction dispatch), so the decryptor has to somehow find the absolute address of the encrypted payload in order to modify it.

The simplest way to derive a pointer to the encrypted payload is to use the `call` instruction. When `call` is executed, 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. This technique is used by the decryptor shown in Figure 1. The encrypted payload begins at addresses `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`. Thus, 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, starting from the last encrypted byte.

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 [37]. The stored record includes the instruction pointer of the FPU, thus if a floating point instruction has been executed as part of the decryptor, `fstenv` can be used to retrieve its absolute memory address.

A third getPC technique is possible by exploiting the structured exception handling (SEH) mechanism of Windows [38]. However this technique is feasible 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 (cf. Section 5.2), only Alpha2 supports this type of getPC, although not by default.

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

Known operand values. Polymorphic shellcode engines produce generic decryptor code for a specific hardware platform, which 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. For instance, going back to Fig. 3, the execution trace of the Countdown decoder is always the same, independently of the process in which it has been injected. Indeed, the decoder 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 instructions. The emulator can correctly initialize any registers used, follow stack operations, compute all effective addresses, and even follow self modifications, since every operand eventually becomes known.

Note that, depending on the vulnerability, a skilled attacker may be able to construct a non-self-contained decryptor, which our approach would not be able to fully execute. This can be possible by including in the computations of the decoder values read by known locations of the memory image of the vulnerable process that remain consistent across all vulnerable systems. We further discuss this issue in Section 6.

4.2 Detection Algorithm

The algorithm takes as input a byte stream, such as a reassembled TCP stream or the payload of a UDP packet, captured passively from the network, and reasons whether it contains polymorphic shellcode. The algorithm executes the input byte stream 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 pseudocode of the algorithm is presented in Figure 4 with several simplifications for brevity. Each input buffer is mapped to a random location in the virtual address space of the emulator. This is similar 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 for 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.

Running the shellcode. The main routine, `emulate`, takes as parameters the address and the length of the input stream. Depending on the vulnerability, the injected code may be located in 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 the 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  }
```

Fig. 4. Simplified pseudo-code for the detection algorithm.

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 [7, 8], 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. 3, 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, as discussed in the following section.

For large input streams, starting a new execution from each and every position incurs a very high execution overhead per stream. We have implemented the following optimization in order to mitigate this effect. Since injected shellcode is treated by the vulnerable application as a string, any NULL byte in the shellcode will truncate it and render it nonfunctional. For this reason, shellcodes cannot contain a zero byte. We exploit this restriction by taking advantage of the zero bytes found in binary network traffic. Before starting the execution from position i, a look-ahead scan is performed to find the first zero byte after byte i. If a zero byte is found at position j, and $j - i$ is less than a minimum size S, then the positions from i to j are skipped and the algorithm continues from position $j + 1$. We have chosen a rather conservative value for $S = 50$, given that most polymorphic shellcodes have a size greater than 100 bytes.

For each position `pos`, the algorithm enters the main simulation loop (line 6), in which a new instruction is fetched, decoded, and executed. 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 different execution chain of the same input buffer or as part of a loop body in the same execution chain, the instruction is instantly fetched from the translation cache.

Detection heuristic. Although the execution behavior of random binary code is undefined, there exists a generic execution pattern inherent to all polymorphic shellcodes that allows us to accurately distinguish polymorphic code injection attacks from normal requests. During decryption, the decoder 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 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 happen rarely during the execution of random code.

Only instructions that have an operand whose data is located in memory 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. Given that input streams are mapped to a random memory location and that before each execution the CPU registers, some of which usually take part in the computation of the effective address, are randomized, the probability to encounter a memory read from the input buffer in random code is low. In contrast, the decryptor will access tens or hundreds of *different* memory locations within the input buffer. This observation led us to initially choose as the detection criterion the number of reads from distinct memory locations of the input buffer. For the sake of brevity, we refer to memory reads from distinct locations of the input buffer as *"payload reads."* When the number of payload reads for a given execution chain reaches a certain payload reads threshold (PRT), this is an indication of 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 shellcode. However, preliminary experiments with network traces revealed 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 with a memory operand that happened to read hundreds of different memory locations from the input buffer. We addressed this issue by defining a more strict detection criterion. As discussed in Section 4.1, a mandatory operation of every polymorphic shellcode is to find its location in memory 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. We discuss in detail the detection criterion and its effectiveness in terms of false positives in Section 5.1. The experimental evaluation showed that the above heuristic allows for accurate detection of polymorphic shellcode with zero false positives. Another option for enhancing the detection heuristic would be to look for linear payload reads from a contiguous memory region. However, this heuristic can be tricked by spliting the encrypted payload into nonadjacent parts and decrypting it in a random order [39].

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

Invalid instruction. The execution stops 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 execution of shellcode. 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 will bypass them during execution—otherwise the execution would fail. In that case, the emulator will also follow the real execution, so such instructions will not cause any inconsistensy. At the same time, privileged or illegal instructions appear relatively often in random data, helping this way to distinguish between benign requests and attack vectors.

Invalid memory location. Normally, during the execution of the decoder, the instruction pointer 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" some instructions on the fly, in a similar way to the self-modifications presented in Section 3.2. 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 [40]. Although it is generally difficult to know in advance the contents of a certain memory location, it is easier to find virtual memory addresses that are always mapped into the address space of the vulnerable process. For the same reason, the emulator cannot execute instructions that access unknown memory locations, since the memory contents are not available. Such instructions are ignored and the execution continues normally. However, the emulator keeps track of any memory locations outside of the input buffer that have been written during execution, and marks them as valid memory locations where useful data or code may have been placed. If at any time the program counter (PC) points to such an address, the execution continues normally from that location. In contrast, if the PC points to an address outside the input buffer that has not been written during the particular execution, the execution stops (line 15). In random binary code, this usually happens when the PC reaches the end of the input buffer.

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 "endless" or infinite loops. In such cases, an execution threshold (XT) is necessary to avoid extensive performance degradation or execution hang ups (line 18).

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, as discussed in Section 5.3. 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.

To further mitigate the effect of endless loops, we have implemented a heuristic method for identifying and stopping the execution of infinite loops that may occur in

```
...
0A40   xor ch,0xc3
0A43   imul dx,[ecx],0x5
0A48   mov eax,0xf4
0A4D   jmp short 0xa40
...
```
(a)

```
...
0F30   ror ebx,0x9
0F33   stc
0F34   mov al,0xf4
0F36   jpe 0xf30      ;PF=1
...
```
(b)

Fig. 5. Infinite loops in random code due to (a) unconditional and (b) conditional branches.

random code. Loops are detected dynamically using the method proposed by Tubella et al. [41]. This technique detects the beginning and the termination of iterations and loop executions in run-time using a Current Loop Stack that contains all loops that are being executed at a given time. The following infinite loop cases are detected: (i) there is an unconditional backward branch from address S to address T and there is no control transfer instruction in the range [T,S] (the loop body), and (ii) there is a conditional backward branch from address S to address T, none of the instructions in the range [T,S] is a control transfer instruction, and none of these instructions affects the status flag(s) of the EFLAGS register on which the conditional branch depends on. Examples of the two cases are presented in Figure 5. In example (b), when control reaches the `ror` instruction, the parity flag (PF) flag has been set, as a result of some previous instruction. Since none of the instructions in the loop body affects PF, its value will not change until the execution reaches the jump-if-parity instruction at address `0x0f38`, which will jump back to the `ror` instruction, resulting to an infinite loop.

Clearly, these are very simple cases and more complex infinite loop structures may arise. Our experiments have shown that, depending on the monitored traffic, above heuristics prune about 2-4% of the instruction chains that stop running because they exceed the execution threshold. Usually loops in random code are not infinite, but need to execute for many iterations until completion. Thus, the runtime overhead of any more elaborate infinite loop detection method will be higher than the overhead of simply running the extra infinite loops that may arise until the execution threshold is reached.

4.3 Implementation

In this section we describe our prototype implementation of the network-level detector. The detector passively captures the monitored network traffic using `libpcap` [42] and reassembles TCP/IP streams using `libnids` [43]. The input buffer size is set to 64KB, which is more than enough for typical service requests. Especially for web traffic, HTTP/1.1 pipelined requests trhough persistent connections are split to separate input streams, otherwise an attacker could evade detection by filling the stream with benign requests until exceeding the buffer size. Instruction set simulation has been implemented interpretively, with a typical fetch, decode, and execute cycle. Accurate instruction decoding, crucial for the identification of invalid instructions, is performed using `libdasm` [44]. For our prototype, we have implemented a subset of the IA-32 instruction set that includes all general-purpose instructions, but no FPU, MMX, SSE, and SSE2 instructions, except `fstenv/fnstenv`, `fsave/fnsave`, and `rdtsc`. However, *all* instructions are fully decoded, and if during execution an unimplemented instruction is encountered, the emulator proceeds normally to the next instruction. The

Service	**Port Number**	**Number of streams**	**Total size**
www	80	1759950	1.72 GB
NetBIOS	137–139	246888	311 MB
microsoft-ds	445	663064	912 MB

Table 1. Characteristics of client-to-server network traffic traces.

implemented subset suffices for the complete and correct execution of all tested shellcodes (cf. Section 5.2). Even the highly obfuscated shellcodes generated by the TAPiON engine [11], which intersperses FPU instructions among the decoder code, are executed successfully, since any FPU instructions are used as NOPs and do not take part in the useful computations of the decoder.

5 Experimental Evaluation

In this section we evaluate the performance of the proposed approach using our prototype implementation. In all experiments, the detector was running on a PC equipped with a 2.53 GHz Pentium 4 processor and 1 GB RAM, running Debian Linux (kernel v2.6.7). For trace-driven experiments, we used full packet traces of traffic from ports related with the most exploited vulnerabilities, captured at ICS-FORTH and the University of Crete. Trace details are summarized in Table 1. Since remote code-injection attacks are performed using a specially crafted request to a vulnerable service, we keep only the client-to-server traffic of network flows. For large incoming TCP streams, e.g., due to a file upload, we keep only the first 64KB. Note that these traces represent a significantly smaller portion of the total traffic that passed by through the monitored links during the monitoring period, since we keep only client-initiated traffic.

5.1 Tuning the Detection Heuristic

We first assess the possibility that the detection algorithm incorrectly detects benign data as polymorphic shellcode. As discussed in Section 4.2, the detection criterion requires the execution of some form of getPC code, followed by a number of reads from distinct memory locations of the input buffer that reaches a certain payload reads threshold. Our initial implementation of this heuristic was the following: if an execution chain contains a `call`, `fstenv`, or `fsave` instruction, followed by PRT or more payload reads, then it belongs to a polymorphic shellcode. The existence of one of the four `call`, two `fstenv`, or two `fsave` instructions of the IA-32 instruction set serves as a simple indication of the potential execution of getPC code. We evaluated this heuristic using the network traces presented in Table 1 as input to the detection algorithm. Only 13 streams were found to contain an execution chain with a `call` or `fstenv` instruction followed by payload reads. In the worst case, there were five payload reads, allowing for a minimum value for PRT = 6. However, since the false positive rate is a crucial factor for the applicability of our detection method, we further explored the quality of the detection heuristic using a significantly larger data set.

Payload Reads	**Streams**			
	Initial Heuristic		**Improved Heuristic**	
	#	%	#	%
1	409	0.02045	22	0.00110
2	39	0.00195	5	0.00025
3	10	0.00050	3	0.00015
4	9	0.00045	1	0.00005
5	3	0.00015	1	0.00005
6	5	0.00025	1	0.00005
7–100	44	0.00220	**0**	**0**
100–416	37	0.00185	**0**	**0**

Table 2. Streams that matched the detection heuristic with a given number of payload reads.

We generated two million streams of varying sizes uniformly distributed between 512 bytes and 64 KB with random binary content. The total size of the data set was 61 GB. The results of the evaluation are presented in Table 2, under the column "Initial Heuristic." From the two million streams, 556 had an execution chain that contained a getPC instruction followed by payload reads. Although 475 out of the 556 streams had at most six payload reads, there were 44 streams with tens of payload reads and 37 streams with more than 100 payload reads, reaching 416 in the most extreme case. As we show in Section 5.2, there are polymorphic shellcodes that execute as few as 32 payload reads. As a result, PRT cannot be set to a value greater than 32 since it would otherwise miss some polymorphic shellcodes. Thus, the above heuristic incorrectly identifies these cases as polymorphic shellcode.

Although only the 0.00405 % of the total streams resulted to a false positive, we can devise an even more strict criterion to further lower the false positive rate. Payload reads occur in random code whenever the memory operand of an instruction accidentally refers to a location within the input buffer. In contrast, the decoder of a polymorphic shellcode explicitly refers to the memory region of the encrypted payload based on the value of the instruction pointer that is pushed in the stack by a `call` instruction, or stored in the memory location specified in an `fstenv` instruction. Thus, after the execution of such an instruction, the next mandatory step of a getPC code is to read the instruction pointer from the memory location where it was stored. This led us to further enhance the detection criterion as follows: *if an execution chain contains a* `call`*,* `fstenv`*, or* `fsave` *instruction, followed by a read from the memory location where the instruction pointer was stored as a result of one of the above instructions, followed by PRT or more payload reads, then it belongs to a polymorphic shellcode.*

Using the same data set, the enhanced criterion results to significantly fewer matching streams, as shown under the column "Enhanced Heuristic" of Table 2. In the worst case, one stream had an execution chain with a `call` instruction, an accidental read from the memory location of the stack where the return address was pushed, and six payload reads, which allows for a lower bound for PRT = 7.

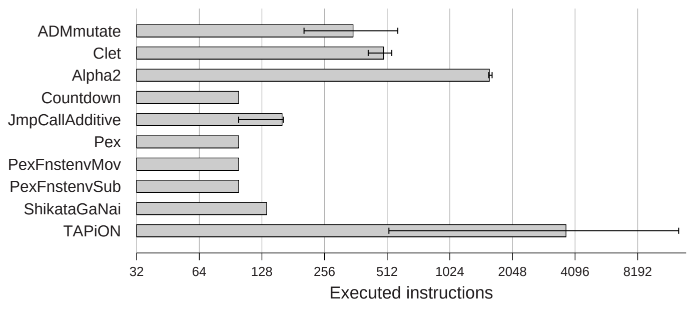

Fig. 6. Average number of executed instructions for the complete decryption of an 128 byte shellcode encrypted using several polymorphic and encryption engines. Bars correspond to the average among 1000 samples for each engine. The ends of the range bars correspond to the samples with the minimum and maximum number of executed instructions.

5.2 Validation

Polymorphic Shellcode Execution. We tested the capability of the emulator to correctly execute polymorphic shellcode using real samples produced by off-the-shelf polymorphic shellcode engines. We generated mutations of an 128 byte shellcode using the Clet [16], ADMmutate [15], and TAPiON [11] polymorphic shellcode engines, and the Alpha2, Countdown, JmpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShigataGaNai shellcode encryption engines of the Metasploit Framework [34]. TAPiON—the most recent of the engines—produces highly obfuscated code using anti-disassembly and anti-emulator techniques, many garbage instructions, code block transpositions, and on-the-fly instruction generation. Specifically, in some cases, the decryptor produces on-the-fly some code in the stack, jumps to it, and then jumps back to the original decryptor code. For each engine, we generated 1000 instances of the original shellcode. For engines that support options related to the obfuscation degree, we split the 1000 samples evenly using all posible parameter combinations. The execution of each sample stops when the complete original shellcode is found in the memory image of the emulator.

Figure 6 shows the average number of executed instructions that required for the complete decryption of the payload for each engine. The ends of range bars, where applicable, correspond to the samples with the minimum and maximum number of executed instructions. In all cases, the emulator decrypts the original shellcode correctly. Figure 7 shows the average number of payload reads for the same experiment. For simple encryption engines, the decoder decrypts four bytes at a time, resulting to 32 payload reads. On the other extreme, shellcodes produced by the Alpha2 engine perform more that 500 payload reads. Alpha2 produces alphanumeric shellcode, and consequently, the shellcode uses a considerably smaller subset of the IA-32 instruction set, which forces it to execute much more instructions in order to achieve the same goals. Given that 128 bytes is a rather small size for a functional polymorphic shellcode, these results can be used to derive an indicative upper bound for PRT = 32. Combined with the results of the previous section, this allows for a range of possible values for PRT from 7 to 31.

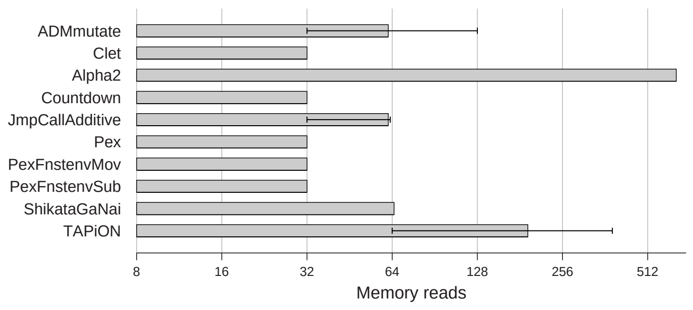

Fig. 7. Average number of payload reads for the complete decryption of an 128 byte shellcode encrypted using several polymorphic and encryption engines. Bars correspond to the average among 1000 samples for each engine. The ends of the range bars correspond to the samples with the minimum and maximum number of payload reads.

For our experiments we choose for PRT the median value of 19, which allows for even more increased resilience to false positives.

Detection Effectiveness. To test the efficacy of our detection method, we launched a series of remote code-injection attacks using the Metasploit Framework [34] against an unpatched Windows XP host running Apache v1.3.22. Attacks were launched from a Linux host using Metsploit's exploits for the following vulnerabilities: Apache win32 chunked encoding [45], Microsoft RPC DCOM MS03-026 [46], Microsoft LSASS MS04-011 [47]. The detector was running on a third host that passively monitored the incoming traffic of the victim host. For the payload we used the `win32_reverse` shellcode, which connects back to the attacking host and spawns a shell, encrypted with different engines. We tested all combinations of the three exploits with the engines presented in the previous section. All attacks were detected successfully, with zero false negatives.

5.3 Processing Cost

We evaluate the raw processing speed of our prototype implementation using the network traces presented in Table 1. Although emulation is a CPU-intensive operation, we show that it is feasible to apply it for network-level polymorphic attack detection. One of the main factors that affects the processing speed is the execution threshold XT beyond which an execution chain stops. The larger the XT, the more the time spent on streams with long execution chains. As shown in Figure 8, as XT increases, the throughput decreases, especially for ports 139 and 445. The reason for the linear decrease of the throughput for these ports is that some streams have very long execution chains that always reach the XT, even when it is set to large values. As XT increases, the emulator spends even more cycles on these chains, which increases the overall throughput.

We further explore this effect in Figure 9, which shows the percent of streams with an execution chain that reaches a given execution threshold. As XT increases, the number of streams that reach it decreases. This effect is occurs only for low XT values

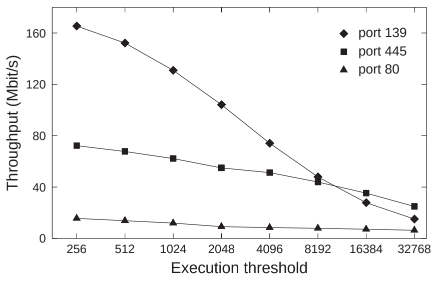

Fig. 8. Processing speed for different execution thresholds.

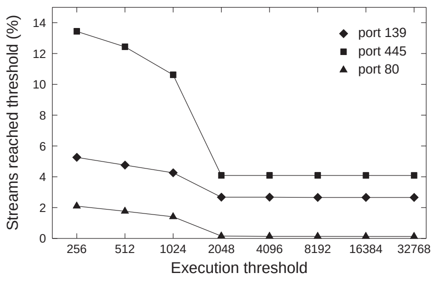

Fig. 9. Percent of streams that reach the execution threshold.

due to large code blocks with no branch instructions that are executed linearly. For example, the execution of blocks that have more than 256 but less than 512 valid instructions, reach a threshold of 256, but fully execute with a threshold of 512. However, the occurrence probability of such blocks is reversely proportional to their length, due to the illegal or privileged instructions that accidentally occur in random code. Thus, the percent of streams that reach XT stabilizes beyond the value of 2048, and is solely due to execution chains with endless loops, which usually require a prohibitive number of instructions in order to complete. In contrast, port 80 behaves differently because the ASCII data that dominate in web requests produce mainly forward jumps, and thus the occurence of endless loops rare. Therefore, beyond an XT of 2048, the percent of streams with an execution chain that stops due to the threshold is negligible, reaching 0.12%. However, since ASCII web requests do not contain any null bytes, the zero-delimited chunks optimization does not reduce the number of execution chains per stream, which results to a lower processing speed.

Figures 8 and 9 represent two conflicting tradeoffs related to the execution threshold. Presumably, the higher the processing speed, the better, which leads towards lower XT values. On the other hand, as discussed in Section 4.2, it is desirable to have as few streams with execution chains that reach XT as possible, i.e., a higher XT, which will increase the visibility of endless loop attacks. Based on the second requirement, XT values higher than 2048 do not offer any improvement to the percent of streams that reach it, which stabilizes at 2.65% for port 139 and 4.08% for port 445.

At the same time, an execution threshold of 2048 allows for a quite decent processing speed, especially when taking into account that live incoming traffic will usually have relatively lower volume than the monitored link's bandwidth, particularly if the services are not related to file uploads. We should also stress that our prototype is highly unoptimized. For instance, a threaded code [48] emulator combined with optimizations such as lazy condition code evaluation [49] would result to better performance.

A final parameter that we should take into account is whether the execution threshold allows polymorphic shellcodes to perform enough payload reads to reach the payload reads threshold and be successfully detected. As shown in Section 5.2, the complete decryption of some shellcodes requires the execution of even more than 10000 instructions, which is much higher than an XT as low as 2048. However, as shown in

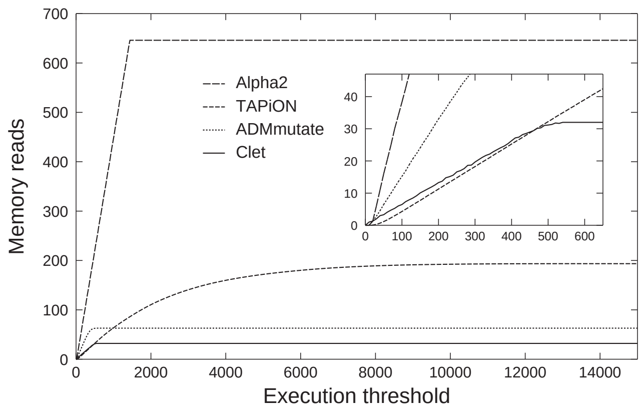

Fig. 10. The average number of payload reads from Figure 7 that a given execution threshold allows to be executed.

Figure 10, even lower XT values, which give better throughput for binary traffic, allow for the execution of more than enough payload reads. For example, in all cases, the chosen PRT value of 19 is reached by executing only 300 instructions.

6 Limitations

A fundamental limitation of our method is that it detects only polymorphic shellcodes that decrypt their body before executing their actual payload. Plain or completely metamorphic shellcodes that do not perform any self-modifications are not captured by our detection heuristic. However, we have yet to see a purely metamorphic shellcode engine implementation, while polymorphic engines are becoming more prevalent and complex [11], mainly for two reasons. First, polymorphic shellcode is increasingly used for evading detection. Second, the ever increasing functionality of recent shellcodes makes their construction more complex, while at the same time their code should not contain NULL and, depending on the exploit, other restricted bytes. Thus, it is easier for shellcode authors to avoid such bytes in their code by encoding their body using an off-the-shelf encryption engine, rather than having to handcraft the shellcode [1]. In many cases the latter is non-trivial, since many exploits require the avoidance of many different restricted bytes [34], with the most extreme cases requiring purely ASCII shellcode [50].

Our method works only with self-contained shellcode. Although current polymorphic shellcode engines produce self-contained code, a motivated attacker could hinder network-level emulation by constructing a shellcode that involves registers or memory locations with a priori known values that remain constant across all vulnerable systems. However, the extended use of hardcoded addresses results in more fragile code [1], especially as address space randomization methods are becoming more prevalent [51]. In our future work we plan to explore the applicability of our approach at the host-level, where the emulator would have access to the address space of the vulnerable process.

Another possible evasion method is the placement of endless loops for reaching the execution threshold before any actual decryptor code runs. Although this is a well-known problem in the context of virus scanners for years, if attackers start to employ such evasion techniques, our method will still be useful as a first-stage anomaly detector for application-aware NIDS like shadow honeypots [52], given that the appearance of endless loops in random code is rare, as shown in Section 5.3.

Finally, unicode-proof shellcodes [40], which become functional after being transformed according to the unicode encoding, would not be executed correctly by our current prototype. This is an orthogonal problem that can be solved by augmenting the emulator with appropriate filters that transform the input streams in a similar way to the protected service.

7 Conclusion

We have considered the problem of detecting polymorphic shellcode injection attacks at the network level. The main question asked by us, as well as other researchers, is whether such attacks can be identified purely based on the limited information available through network traffic monitoring.

The starting point for our work is the observation that previous proposals that rely on static analysis are insufficient, because they can be bypassed using techniques such as simple self-modifications. In response to this observation, we set out to explore the feasibility of performing more accurate analysis through network level execution of potential shellcodes by employing a fully-blown processor emulator on the NIDS side. We have examined the execution profiles of a large number of shellcode generators and identified properties that can distinguish shellcodes from normal traffic with reasonable accuracy. Our analysis indicates that our approach can detect all known classes of polymorphic shellcodes, including those that employ certain forms of self-modification which are not detected by previous proposals. Furthermore, our experiments suggest that the cost of our approach is modest.

However, further analysis on the robustness of our approach also revealed that attackers can succeed in circumventing our techniques if the shellcode is not self-contained. In particular, the attacker can leverage context not available at the network level for building shellcodes that cannot be unambiguously executed on the network level processor emulator. Detecting such attacks remains an open problem.

One way of tackling this problem is to feed the necessary host-level information to the NIDS as suggested in [53], but the feasibility of doing so is yet to be proven. A major concern is that, in most cases, bypassing shellcode detection techniques, including our own, has been relatively straightforward, and appears to carry no additional cost or risks for the attacker. Thus, these techniques do not necessarily “raise the bar” for the attacker, while their cost for the defender in terms of the resources that need to be devoted to detection can be significant. At this point it remains unclear whether network level detection is feasible. Nevertheless, we believe that the work described in this report brings us one step closer to answering this question.

References

[1] sk, "History and advances in windows shellcode," *Phrack*, vol. 11, no. 62, July 2004.

[2] H.-A. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proceedings of the* 13^{th} *USENIX Security Symposium*, 2004, pp. 271–286.

[3] S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proceedings of the* 6^{th} *Symposium on Operating Systems Design & Implementation (OSDI)*, Dec. 2004.

[4] 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.

[5] 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.

[6] K. Wang and S. J. Stolfo, "Anomalous Payload-based Network Intrusion Detection," in *Proceedings of the* 7^{th} *International Symposium on Recent Advanced in Intrusion Detection (RAID)*, September 2004, pp. 201–222.

[7] 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.

[8] 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.

[9] P. Ször and P. Ferrie, "Hunting for metamorphic," in *Proceedings of the Virus Bulletin Conference*, Sept. 2001, pp. 123–144.

[10] 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.

[11] P. Bania, "TAPiON," 2006, http://pb.specialised.info/all/tapion/.

[12] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of USENIX LISA '99*, November 1999, (software available from *http://www.snort.org/*).

[13] V. Paxson, "Bro: A system for detecting network intruders in real-time," in *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[14] C. Jordan, "Writing detection signatures," *USENIX ;login:*, vol. 30, no. 6, pp. 55–61, December 2005.

[15] K2, "ADMmutate," 2006, http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[16] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk, "Polymorphic shellcode engine using spectrum analysis," *Phrack*, vol. 11, no. 61, Aug. 2003.

[17] 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.

[18] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis, "STRIDE: Polymorphic Sled Detection through Instruction Sequence Analysis," in *Proceedings of the* 20^{th} *IFIP International Information Security Conference (IFIP/SEC)*, June 2005.

[19] J. R. Crandall, S. F. Wu, and F. T. Chong, "Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities," in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2005.

[20] A. Pasupulati, J. Coit, K. Levitt, S. Wu, S. Li, J. Kuo, and K. Fan, "Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities," in *Proceedings of the Network Operations and Management Symposium (NOMS)*, April 2004, pp. 235–248.

[21] C. Kreibich and J. Crowcroft, “Honeycomb – creating intrusion detection signatures using honeypots,” in *Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II)*, Nov. 2003.

[22] O. Kolesnikov, D. Dagon, and W. Lee, “Advanced polymorphic worms: Evading IDS by blending in with normal traffic,” College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, 2004, http://www.cc.gatech.edu/~ok/w/ok_pw.pdf.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] C. S. Collberg and C. Thomborson, “Watermarking, tamper-proffing, and obfuscation: tools for software protection,” *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[28] 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.

[29] 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. 2006, pp. 194–206.

[30] B. Schwarz, S. Debray, and G. Andrews, “Disassembly of executable code revisited,” in *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE)*, 2002.

[31] 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.

[32] 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.

[33] F. B. Cohen, “Operating system protection through program evolution,” *Computer and Security*, vol. 12, no. 6, pp. 565–584, 1993.

[34] “Metasploit Project,” 2006, http://www.metasploit.com/.

[35] C. Cifuentes and K. J. Gough, “Decompilation of binary programs,” *Software—Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[36] G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” in *Proceedings of the International Conference on Compiler Construction (CC)*, Apr. 2004.

[37] Noir, “GetPC code (was: Shellcode from ASCII),” June 2003, http://www.securityfocus.com/archive/82/327100/2006-01-03/1.

[38] C. Ionescu, “GetPC code (was: Shellcode from ASCII),” July 2003, http://www.securityfocus.com/archive/82/327348/2006-01-03/1.

[39] F. Perriot, P. Ferrie, and P. Ször, “Striking similarities,” *Virus Bulletin*, pp. 4–6, May 2002.

[40] Obscou, “Building ia32 ’unicode-proof’ shellcodes,” *Phrack*, vol. 11, no. 61, Aug. 2003.

[41] J. Tubella and A. González, “Control speculation in multithreaded processors through dynamic loop detection,” in *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA)*, 1998.

[42] S. McCanne, C. Leres, and V. Jacobson, “Libpcap,” 2006, http://www.tcpdump.org/.

[43] R. Wojtczuk, “Libnids,” 2006, http://libnids.sourceforge.net/.

[44] jt, “Libdasm,” 2006, http://www.klake.org/~jt/misc/libdasm-1.4.tar.gz.

[45] “Apache Chunked Encoding Overflow,” 2002, http://www.osvdb.org/838.

[46] “Microsoft Windows RPC DCOM Interface Overflow,” 2003, http://www.osvdb.org/2100.

[47] “Microsoft Windows LSASS Remote Overflow,” 2004, http://www.osvdb.org/5248.

[48] J. R. Bell, “Threaded code,” *Comm. of the ACM*, vol. 16, no. 6, pp. 370–372, 1973.

[49] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[50] Rix, “Writing ia32 alphanumeric shellcodes,” *Phrack*, vol. 11, no. 57, Aug. 2001.

[51] S. Bhatkar, D. C. DuVarney, and R. Sekar, “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits,” in *Proceedings of the* 12^{th} *USENIX Security Symposium*, 2003.

[52] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis, “Detecting Targeted Attacks Using Shadow Honeypots,” in *Proceedings of the* 14^{th} *USENIX Security Symposium*, August 2005, pp. 129–144.

[53] H. Dreger, C. Kreibich, V. Paxson, and R. Sommer, “Enhancing the accuracy of network-based intrusion detection with host-based context,” in *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2005.