J Comput Virol (2007) 2:257–274
DOI 10.1007/s11416-006-0031-z

ORIGINAL PAPER

Network-level polymorphic shellcode detection using emulation

Michalis Polychronakis · Kostas G. Anagnostakis · Evangelos P. Markatos

Received: 31 July 2006 / Accepted: 1 November 2006 / Published online: 23 December 2006

Abstract Significant progress has been made in recent years towards preventing code injection attacks at the network level. However, as state-of-the-art attack detection technology becomes more prevalent, attackers are likely to evolve, employing techniques such as polymorphism and metamorphism to defeat these defenses. A major outstanding question in security research and engineering is thus whether we can proactively develop the tools needed to contain advanced polymorphic and metamorphic attacks. While recent results have been promising, most of the existing proposals can be defeated using only minor enhancements to the attack vector. In fact, some publicly-available polymorphic shellcode engines are currently one step ahead of the most advanced publicly-documented network-level detectors. In this paper, we present a heuristic detection method that scans network traffic streams for the presence of previously unknown polymorphic shellcode. In contrast to previous work, our approach relies on a NIDS-embedded CPU emulator that executes every potential instruction sequence in the inspected traffic, aiming to identify the execution behavior of polymorphic shellcode. 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.

M. Polychronakis (✉) · E. P. Markatos
Institute of Computer Science,
Foundation for Research & Technology – Hellas,
Heraklion, Crete, Greece
e-mail: mikepo@ics.forth.gr

E. P. Markatos
e-mail: markatos@ics.forth.gr

K. G. Anagnostakis
Institute for Infocomm Research,
Singapore, Singapore
e-mail: kostas@i2r.a-star.edu.sg

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 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 [1]. 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 paper 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 [2–10]. 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 [11], 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 [12], which collectively refers to techniques such as dead-code insertion, code transposition, register reassignment, and instruction substitution [13], 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 automatic containment of advanced polymorphic attacks at the network-level. 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, most of the existing proposals can be easily defeated.

In this paper, we revisit the question of whether polymorphic shellcode is detectable at the network-level. We present a detection heuristic that tests byte sequences in network traffic for properties similar to polymorphism. Specifically, we speculatively execute potential instruction sequences and compare their execution profile against the behavior observed to be inherent to polymorphic shellcode. 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 paper is organized as follows. We summarize related work in Sect. 2, and discuss techniques that can be used for evading current network-level detectors in Sect. 3. We present in detail our emulation-based detection method in Sect. 4, and experiments examining the performance of our approach in Sect. 5. Finally, Sect. 6 discusses limitations and issues that need further research, and Sect. 7 concludes the paper.

2 Related work

Network intrusion detection systems (NIDS) like Snort [14] and Bro [15] have been extensively used for shellcode detection. Although such systems usually detect known attacks, for which a precise signature exists, they can also be used for detecting previously unseen attacks using generic signatures that match components common to similar exploits, such as the NOP sled, protocol framing, or specific parts of the shellcode [16]. As a response to signature-based NIDS, attackers have started to employ encryption and polymorphism for evading detection [17–20].

Initial approaches on zero-day polymorphic shellcode detection focused on the identification of the sled component [21,22] that is often prepended to the beginning of the shellcode. 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 [23]. In fact, most infamous Internet worms so far did not employ sleds. Our approach focuses on the detection of the polymorphic shellcode itself, and thus works even in the absence of a sled component. Buttercup [24] attempts to detect polymorphic buffer overflow attacks by identifying the ranges of the possible return addresses for existing buffer overflow vulnerabilities. Unfortunately, this heuristic cannot be employed against some of the more sophisticated buffer overflow attack techniques [25].

Several research efforts have focused on the automated generation of signatures for previously unknown worms. These methods are based on the prevalence of common byte sequences across different worm instances, among other characteristics, and derive signatures that match zero day worms by correlating payloads from different suspicious traffic flows [2,3,26]. However, these approaches are prone to false positives, and are ineffective against polymorphic worms [27], which do not contain sufficiently long common byte sequences.

Polygraph [4], PAYL [6], PADS [5], and Hamsa [10], generate signatures that can capture polymorphic worms by identifying common invariants among different worm instances, such as return addresses, protocol framing, and poor obfuscation. The derived signatures are expressed as regular expressions or statistical byte distributions. Although above approaches can identify simple obfuscated worms, their effectiveness is still questionable in the presence of extensive polymorphism [20]. Furthermore, Polygraph and Hamsa rely on an first-level classifier that splits the traffic into two different pools for innocuous and malicious samples, which introduces an additional avenue for evasion [28].

A fundamental limitation of all above automated signature generation methods is that they require multiple worm instances before reasoning for a threat, which makes them ineffective against targeted attacks. In contrast, our proposed method identifies each attack separately, which makes it also effective for 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 inside network flows. Payer et al. [29] 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. 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. SigFree [9] detects the presence of attack code inside network packets by pruning seemingly useless instructions using data flow anomaly detection, and finding an increased number of remaining useful instructions.

The main limitation of above static analysis based approaches, and main motivation for our work, is that an attacker can effectively and effortlessly hinder static analysis, and thus evade detection. We discuss such evasion methods in the following section.

3 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 [7–9,21,22, 29]. These approaches treat the input network stream as potential machine code and analyze it for signs of malicious 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 [21,22] or shellcode [29]. 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 [7–9].

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 [30–32], 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 [33–35].

3.1 Thwarting disassembly

There are two main disassembly techniques: *linear sweep* and *recursive traversal* [36]. 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 [37]. 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 [38]. 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 [39].

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.

Figure 1 shows the disassembly of the decoder part of a shellcode encrypted using the Countdown encryption engine of the Metasploit Framework [40] using linear sweep and recursive traversal. The code has been

```
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)

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

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 [36, 38, 41, 42]. 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.

3.2 Thwarting control and data flow 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. [7] use the CFG of several instances of a polymorphic worm to detect structural similarities between different mutations. Chinchani et al. [8] differentiate between data and exploit code in network streams based on the control flow of the extracted code.

SigFree, proposed by Wang et al. [9], 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

```
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
```

Fig. 2 A modified, static analysis resistant version of the Countdown decoder

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.

A very simple example of this technique, also known as "patching," is presented in Fig. 2, which shows the recursive traversal disassembly of a modified version of the Countdown decoder presented in Fig. 1. 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

```
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
```

Fig. 3 Execution trace of the modified Countdown decoder

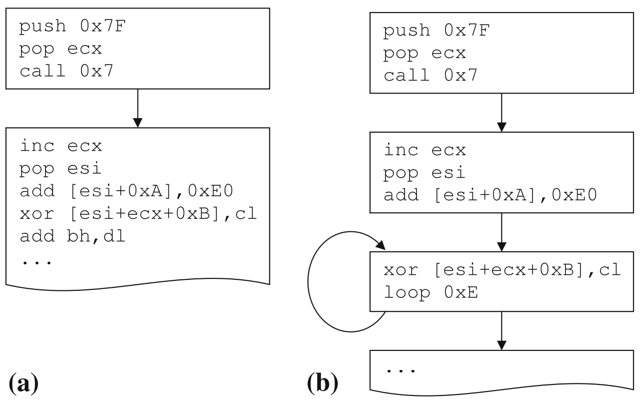

Fig. 4 Control flow graph of the modified Countdown decoder (**a**) based on the code derived using recursive traversal disassembly, and (**b**) based on its actual execution

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. 3.

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, as discussed in Sect. 4.1.2.

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. 4, 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.

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 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, 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. 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.

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 the process being exploited might not be known in advance. At first sight, under these extremely obscure conditions, it does not seem feasible to fully simulate the execution of arbitrary shellcode by relying only to the captured attack vector. For instance, knowing the OS type and version is crucial for emulating system calls.

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.

Up to this point, 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 target 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 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.

4.1.1 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.

4.1.2 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 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 in order to reference the payload memory locations. This technique is used by the decryptor shown in Fig. 1. 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.

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

```
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
...
```

Fig. 5 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 [43]. 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 5 shows the decoder generated by the PexFnstenvMov engine of the Metasploit Framework [40], 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, as presented in Sect. 3.2, 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 [44]. 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 (cf. Sect. 5.2.1), only Alpha2 [45] supports this type of getPC, although not by default.

4.1.3 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. 3 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.

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 could be possible by including in the computations of the decoder existing data that reside at known locations of the memory image of the vulnerable process, and which remain consistent across all vulnerable systems. Such data are not accessible from the network-level emulator, and thus it cannot follow any instructions that manipulate them. We further discuss this issue in Sect. 6.

4.2 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. 6 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. 7. 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.

Fig. 6 Simplified pseudo-code for the detection algorithm

```
emulate(buf_start_addr, buf_len) {
    invalidate_translation_cache();
    for (pos=buf_start_addr; pos<buf_len; ++pos) {
        PC = pos;
        reset_CPU();
        do {
            /* decode instruction if no entry in translation cache */
            if (translation_cache[PC] == NULL)
                translation_cache[PC] = decode_instruction(buf[PC]);
            if (translation_cache[PC] == (ILLEGAL || PRIVILEGED)
                break;
            execute(translation_cache[PC]); /* changes PC */
            if (vmem[PC] == INVALID)
                break;
        }
        while (num_exec++ < XT);
        if (has_getPC_code && (payload_reads >= PRT)
            return TRUE;
    }
    return FALSE;
}
```

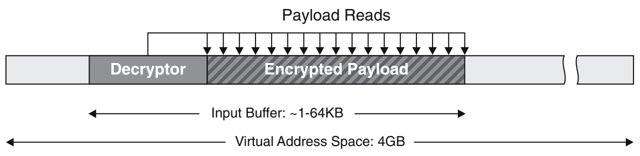

Fig. 7 Memory reads during the decryption of a polymorphic shellcode

4.2.1 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*.

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.

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.

4.2.2 Optimizing performance

For large input streams, starting a new execution from each and every position incurs a high execution overhead per stream. We have implemented the following optimization in order to mitigate this effect. The injected shellcode is treated by the vulnerable application as legitimate input data. Thus, it should conform to any restrictions that input data may have. Since usually the injected code is treated by the vulnerable application as a string, and strings in C are terminated with a NULL byte (a byte with zero value), any NULL byte within the shellcode will truncate it and render the code non-functional. For this reason, the shellcode cannot contain NULL bytes inside its body.

We exploit this restriction by taking advantage of the zero bytes present in binary network traffic. Before starting execution from position *i*, a look-ahead scan is performed to find the first zero byte after position *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.

In the rare case that a protected application accepts NULL characters as part of the input data, this optimization should be turned off. On the other hand, if the application protocol has other restricted bytes, which is quite common [40], extending the above optimization to consider these bytes instead of the zero byte would dramatically improve performance. For instance, the HTTP protocol defines that the request header should be separated from the message body by a CRLF byte combination. Since the two parts of an HTTP request are usually treated separately by web servers, we could extend the above optimization to also consider CRLF byte combinations in case of HTTP traffic.

4.2.3 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. 7, 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. As discussed in Sect. 4.1.2, 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. We discuss in detail this criterion and its effectiveness in terms of false positives in Sect. 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 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 [46].

4.2.4 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.

4.2.4.1 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.

4.2.4.2 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 Sect. 3.2. 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 [47].

Although it is generally difficult to know in advance the contents of a certain memory location, since they usually vary between different systems, 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.

The emulator cannot execute instructions that read unknown memory locations because their contents are not available to the network-level detector. Such instructions are ignored and the execution continues normally. Otherwise, an attacker could trick the emulator by placing NOP-like instructions that read arbitrary data from memory locations known in advance to belong to the address space of the application. However, the emulator keeps track of any memory locations outside of the input buffer that are 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 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, then the execution stops (line 15). In random binary code, this usually happens when the PC reaches the end of the input buffer.

Note that if an attacker knows in advance some memory locations of the vulnerable process that contain code which can be used as part of the shellcode, then the emulator would not be able to fully execute it. We further discuss this issue in Sect. 6.

4.2.4.3 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, as discussed in Sect. 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.

4.2.5 Infinite loop squashing

To further mitigate the effect of seemingly endless loops, we have implemented a heuristic for identifying and stopping the execution of provably infinite loops that may occur in random code. Loops are detected dynamically using the method proposed by Tubella et al.[48]. 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

```
...                                 ...
0A40   xor ch,0xc3                  0F30   ror ebx,0x9
0A43   imul dx,[ecx],0x5            0F33   stc
0A48   mov eax,0xf4                 0F34   mov al,0xf4
0A4D   jmp short 0xa40              0F36   jpe 0xf30       ;PF=1
...                                 ...
          (a)                                 (b)
```

Fig. 8 Infinite loops in random code due to (**a**) unconditional and (**b**) conditional branches

control transfer instruction, and none of the instructions in the range [T,S] affects the status flag(s) of the EFLAGS register on which the conditional branch depends on.

Examples of the two infinite loop cases are presented in Fig. 8. In example (b), when control reaches the `ror` instruction at address `0x0F30`, the parity flag (PF) is already set as a result of some previous instruction. `Ror` affects only the CF and OF flags, `stc` affects only the CF flag, which it sets to 1, and `mov` and `fnop` do not affect any flags. Since none of the instructions in the loop body affects the PF, its value will not change until the jump-if-parity instruction is executed, 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, the above heuristics prune about 3–6% of the execution chains that stop due to reaching the execution threshold. Loops in random code are usually not infinite, but seemingly "endless," being executed for a very large number of 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 they reach the execution threshold.

4.3 Implementation

In this section we describe the prototype implementation of our network-level detector. The detector passively captures network packets using `libpcap` [49] and reassembles TCP/IP streams using `libnids` [50]. The input buffer size is set to 64KB, which is large enough for typical service requests. Especially for web traffic, pipelined HTTP/1.1 requests through persistent connections are split to separate 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, which is crucial for the identification of invalid instructions, is performed using `libdasm` [51]. For our prototype, we have implemented a subset of the IA-32 instruction set, including most of the general-purpose instructions, but no FPU, MMX, SSE, or 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 implemented subset suffices for the complete and correct execution of the decryption part of all the tested shellcodes (cf. Sect. 5.2.1). Even the highly obfuscated shellcodes generated by the TAPiON engine [20], which intersperses FPU instructions among the decoder code, are executed correctly, since the FPU instructions are used only 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 to 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 the client-initiated traffic.

5.1 Tuning the detection heuristic

The major drawback of anomaly-based or heuristics-based attack detection methods is their relatively high false positive ratio. Such methods should have negligible

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

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

false positive ratio in order to be useful. Since the proposed approach is based on a heuristic detection method, we first assess the possibility that the detection algorithm incorrectly detects benign data as polymorphic shellcode.

As discussed in Sect. 4.2.3, the detection criterion requires the execution of some form of getPC code, followed by a number of payload reads that exceed a certain threshold. Our initial implementation of this heuristic was the following: if an execution chain contains a `call`, `fstenv`, `fnstenv`, `fsave`, or `fnsave` instruction, followed by PRT or more payload reads, then it belongs to a polymorphic shellcode. There exist four different versions of the `call` instruction in the IA-32 instruction set. The existence of one of these eight instructions serves just as an indication of the potential execution of getPC code. Only when combined with the execution of several payload reads, it gives a good indication of the execution of a polymorphic shellcode.

5.1.1 Evaluation with real traffic

We evaluated this heuristic using the client-to-server requests from the traces presented in Table 1 as input to the detection algorithm. Only 13 out of the 2,669,902 streams were found to contain an execution chain with a `call` or `fstenv` instruction followed by payload reads, and all of them had non-ASCII content. 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.

5.1.2 Evaluation with synthetic requests

We generated two million streams of varying sizes uniformly distributed between 512 bytes and 64 KB with random binary content. From our experience, binary data is much more likely to give false positives than ASCII only data. 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 payload reads in the most extreme case. As we show in Sect. 5.2.1, 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 shellcodes.

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

Payload	**Streams**			
Reads	**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**

5.1.3 Defining a stricter detection heuristic

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 memory 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 a `call` or `fstenv` instruction, the next mandatory step of a getPC code is to (not necessarily immediately) read the instruction pointer from the memory location where it was stored.

This observation led us to further enhance the detection criterion as follows: *if an execution chain contains one of the eight different* `call`*,* `fstenv`*, or* `fsave` *instructions, 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 heuristic results to significantly fewer matching streams, as shown in Table 2, under the column "Enhanced Heuristic." 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. There were no streams with more than six payload reads, which allows for a lower bound for PRT = 7.

5.2 Validation

5.2.1 *Polymorphic shellcode execution*

We tested the capability of the emulator to correctly execute polymorphic shellcodes using real samples produced by off-the-shelf polymorphic shellcode engines. We generated mutations of an 128 byte shellcode using the Clet [18], ADMmutate [17], and TAPiON [20] polymorphic shellcode engines, and the Alpha2 [45], Countdown, JmpCallAdditive, Pex, PexFnstenvMov, PexFnstenvSub, and ShigataGaNai shellcode encryption engines from the Metasploit Framework [40].

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. In several 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 possible parameter combinations. The execution of each sample stops when the complete original shellcode is found in the memory image of the emulator.

Figure 9 shows the average number of executed instructions that are required for the complete decryption of the payload for the 1000 samples of 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 10 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. ADMmutate decoders read either one or four bytes at a time. On the other extreme, shellcodes produced by the Alpha2 engine perform more than 500 payload reads. Alpha2 produces alphanumeric shellcode using 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 payload, these results can be used to derive an indicative upper bound for PRT = 32 (a higher value would miss such small shellcodes). Combined with the results of the previous section, which showed that the enhanced heuristic is very resilient to accidental payload reads, this allows for a range of possible values for PRT from 7 to 31. For our experiments we choose for PRT the median value of 19, which allows for even

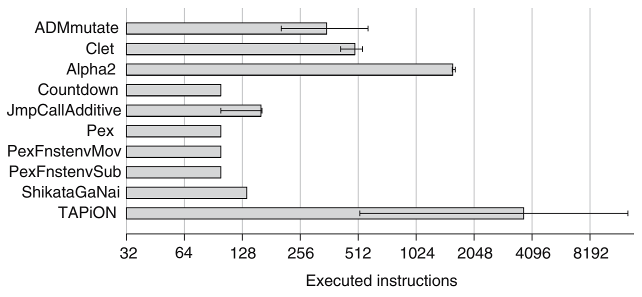

Fig. 9 Average number of executed instructions for the complete decryption of an 128 byte shellcode encrypted using different polymorphic and encryption engines

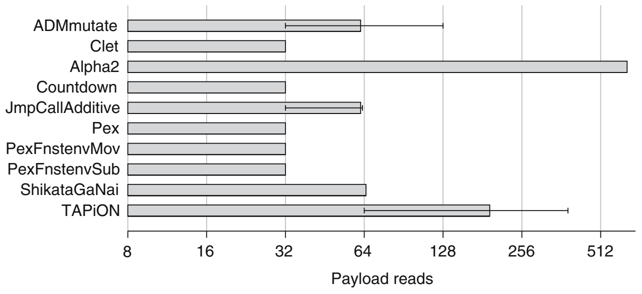

Fig. 10 Average number of payload reads for the complete decryption of an 128 byte shellcode encrypted using different polymorphic and encryption engines

more extreme cases of accidental payload reads not to be misclassified as true positives, while at the same time can capture even smaller shellcodes.

5.2.2 *Detection effectiveness*

To test the efficacy of our detection method, we launched a series of remote code-injection attacks using the Metasploit Framework [40] against an unpatched Windows XP host running Apache v1.3.22. Attacks were launched from a Linux host using Metasploit's exploits for the following vulnerabilities: Apache win32 chunked encoding [52], Microsoft RPC DCOM MS03-026 [53], and Microsoft LSASS MS04-011 [54]. The detector was running on a third host that passively monitored the incoming traffic of the victim host. For the exploit payload we used the shellcode `win32_reverse`, which connects back to the attacking host and spawns a shell, encrypted using 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

In this section we evaluate the raw processing speed of our prototype implementation using the network traces

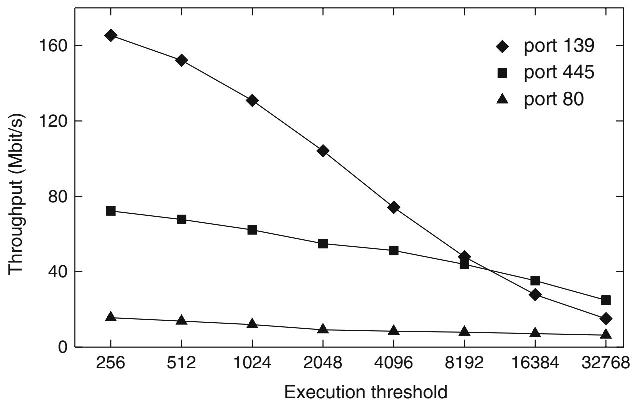

Fig. 11 Processing speed for different execution thresholds

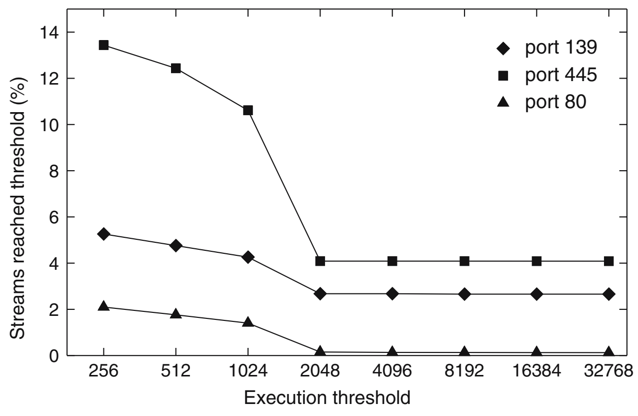

Fig. 12 Percentage of streams that reach the execution threshold

presented in Table 1. Although emulation is a CPU-intensive operation, our aim is to show that it is feasible to apply it for network-level polymorphic attack detection. One of the main factors that affect the processing speed of the emulator is the execution threshold beyond which an execution chain stops. The larger the XT, the more the processing time spent on streams with long execution chains. As shown in Fig. 11, 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. For higher execution thresholds, the emulator spends even more cycles on these chains, which decreases the overall throughput.

We further explore this effect in Fig. 12, which shows the percentage 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 occurs only for low XT values due to large code blocks with no branch instructions that are executed linearly. For example, the execution of linear code blocks with more than 256 but less than 512 valid instructions is terminated before reaching the end when using a threshold of 256, but completes correctly 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 percentage of streams that reach the execution threshold stabilizes beyond the value of 2048. After this value, XT is reached solely due to execution chains with "endless" loops, which usually require a prohibitive number of instructions in order to complete.

In contrast, port 80 traffic behaves differently because the ASCII data that dominate in web requests produce mainly forward jumps, making the occurrence of endless loops extremely rare. Therefore, beyond an XT of 2048, the percentage of streams with an execution chain that stops due to reaching the execution 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.

We should stress at this point that these results refer to the raw processing speed of the detector, which means that under normal operation will be able to inspect traffic of higher speeds, since usually the incoming traffic to some service is less compared to the outgoing traffic. For example, the outgoing traffic from typical web servers is much more than the incoming traffic, because usually the content of web pages is larger than the size of incoming requests. Indeed, a study of the web server traffic at FORTH and the University of Crete for one week showed that from the total traffic, 1.5 % and 14 % was incoming traffic, and 98.5 % and 86 % was outgoing traffic, respectively.

Figures 11 and 12 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 Sect. 4.2.4.3, it is desirable to have as few streams with execution chains that reach the XT as possible. This leads towards higher XT values, which increase the visibility of endless loop attacks. Regarding this second requirement, XT values higher than 2048 do not offer any improvement to the percentage of streams that reach it. After an XT of 2048, the percentage of streams that reach it stabilizes at 2.65% for port 139 and 4.08% for port 445.

At the same time, an XT 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, especially if the protected services are not related

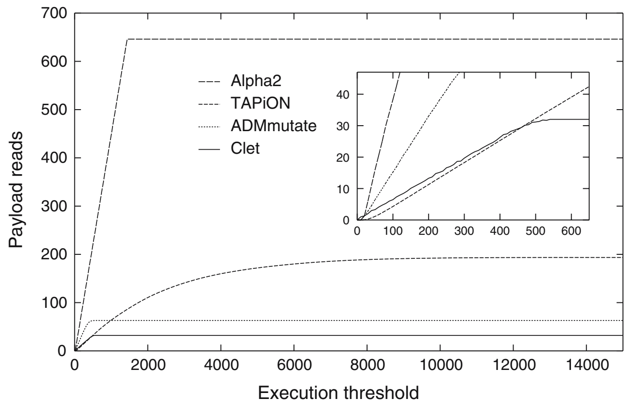

Fig. 13 The average number of payload reads of Fig. 10 that a given execution threshold allows to be executed. All decryptors perform approximately 20 payload reads within the first 300 executed instructions

to file uploads. We should also stress at this point that our prototype is highly unoptimized. For instance, an emulator implemented using threaded code [55], combined with optimizations such as lazy condition code evaluation [56], would result to better performance.

A final issue that we should take into account is to ensure that the selected execution threshold allows polymorphic shellcodes to perform enough payload reads to reach the payload reads threshold and be successfully detected. As shown in Sect. 5.2.1, 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. 13, 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

6.1 Non-self-modifying shellcode

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 [20], mainly for the following two reasons.

First, polymorphic shellcode is increasingly used for evading intrusion detection systems. 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, such as CR, LF, SP, VT, and others. Thus, it is easier for shellcode authors to avoid such bytes in the code by encoding its 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 restricted bytes [40]. There are also cases where even more strict constraints should be handled, such as that the shellcode should survive processing from functions like `toupper()`, or that it should be composed only by printable ASCII characters [19,45].

6.2 Non-self-contained shellcode

Our method works only with self-contained shellcode. Although current polymorphic shellcode engines produce self-contained code, a motivated attacker could evade network-level emulation by constructing a non-self-contained shellcode that involves registers or memory locations with a priori known values that remain constant across all vulnerable systems. For example, if it is known in advance that the address `0x40038EF0` in the vulnerable process' address space contains the instruction `ret`, then the shellcode can be obfuscated by inserting the instruction `call 0x40038EF0` at an arbitrary position in the decoder code. Although this will have no effect to the actual execution of the shellcode, since the flow of control will simply be transferred to address `0x40038EF0`, and from there immediately back to the decoder code, due to the `ret` instruction, the network-level emulator will not execute it correctly, since it cannot follow the jump to address `0x40038EF0`.

However, the extended use of hardcoded addresses results in more fragile code [1], as they tend to change across different software and OS versions, especially as address space randomization schemes are becoming more prevalent [57]. In our future work, we plan to explore ways to augment the network-level detector with host-level information, such as the invariant parts of the address space of the protected processes, in order to make it more robust to such obfuscations.

6.3 Endless loops

Another possible evasion method is the placement of endless loops for reaching the execution threshold before the 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 [58], given that the appearance of endless loops in random code is rare, as shown in Sect. 5.3.

6.4 Transformations beyond the transport layer

Shellcodes contained in compressed HTTP/1.1 connections, or unicode-proof shellcodes [47], which become functional after being transformed according to the unicode encoding by the attacked service, are not executed correctly by our prototype. This is an orthogonal issue that can be addressed by reversing the encoding used in each case by the protected service through appropriate filters before the emulation stage.

Generally, network data that are being transformed above the transport layer, before reaching the core application code, cannot always be effectively inspected using passive network monitoring, as for example in case of encrypted SSL or HTTPS connections. In such cases, our technique can still be applied by moving it from the network-level to a proxy that first decrypts the traffic before scanning it [59]. Another option is to integrate the detector to the end hosts, either at the socket level, by intercepting calls that read network input trough library interposition [60], or at the application level as an extension to the protected service, e.g., as module for the Apache web server [9].

7 Conclusion

We have considered the problem of detecting previously unknown polymorphic code injection attacks at the network level. The main question is whether highly obfuscated versions of such attacks can be identified purely based on the limited information available through passive 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 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 shellcodes produced using various polymorphic shellcode engines, and identified properties that can distinguish polymorphic 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-modifications that 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 [61], 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 accurate network level detection is feasible. Nevertheless, we believe that the work described in this paper brings us one step closer to answering this question.

Acknowledgment This work was supported in part by the projects CyberScope, EAR, and Miltiades, funded by the Greek General Secretariat for Research and Technology under contract numbers PENED 03ED440, USA-022, 05NON-EU-109, respectively, and by the FP6 project NoAH funded by the European Union under contract number 011923. Michalis Polychronakis and Evangelos P. Markatos are also with the University of Crete.

References

1. sk, History and advances in windows shellcode. Phrack **11**(62), (2004)
2. Kim, H.-A., Karp, B.: Autograph: toward automated, distributed worm signature detection. In: Proceedings of the 13th USENIX Security Symposium, pp. 271–286, (2004)
3. Singh, S., Estan, C., Varghese, G., Savage, S.: Automated worm fingerprinting. In: Proceedings of the 6th Symposium on Operating Systems Design & Implementation (OSDI), (2004)
4. Newsome, J., Karp, B., Song, D.: Polygraph: automatically Generating signatures for polymorphic worms. In: Proceedings of the IEEE Security & Privacy Symposium, pp. 226–241, (2005)
5. Tang, Y., Chen, S.: 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. Wang, K., Stolfo, S.J.: Anomalous payload-based network intrusion detection. In: Proceedings of the 7th International Symposium on Recent Advanced in Intrusion Detection (RAID), pp. 201–222, (2004)

7. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), (2005)
8. Chinchani, R., Berg, E.V.D.: 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), (2005)
9. Wang, X., Pan, C.-C., Liu, P., Zhu, S.: Sigfree: a signature-free buffer overflow attack blocker. In: Proceedings of the USENIX Security Symposium (2006)
10. Li, Z., Sanghi, M., Chen, Y., Kao, M.-Y., Chavez, B.: Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, pp. 32–47, 2006
11. Ször, P.: The art of computer virus research and defense. Addison-Wesley Professional, (2005)
12. Ször, P., Ferrie, P.: Hunting for metamorphic. In: Proceedings of the virus bulletin conference. pp. 123–144, (2001)
13. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium (Security'03), (2003)
14. Roesch, M.: Snort: lightweight intrusion detection for networks. In: Proceedings of USENIX LISA '99, November 1999, (software available from http://www.snort.org/)
15. Paxson, V.: Bro: a system for detecting network intruders in real-time. In: Proceedings of the 7th USENIX Security Symposium, (1998)
16. Jordan, C.: Writing detection signatures. USENIX Login **30**(6), 55–61 (2005)
17. K2, ADMmutate, http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz, (2001)
18. Detristan, T., Ulenspiegel, T., Malcom, Y., Underduk, M.: Polymorphic shellcode engine using spectrum analysis. Phrack **11**(61), (2003)
19. Rix, Writing IA32 alphanumeric shellcodes. Phrack **11**(57), (2001)
20. Bania, P.: TAPiON, http://pb.specialised.info/all/tapion/, (2005)
21. Toth, T., Kruegel, C.: Accurate buffer overflow detection via abstract payload execution. In: Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID), (2002)
22. Akritidis, P., Markatos, E.P., Polychronakis, M., Anagnostakis, K.: STRIDE: Polymorphic sled detection through instruction sequence analysis. In: Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC), (2005)
23. Crandall, J.R., Wu, S.F., Chong, F.T.: 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), (2005)
24. Pasupulati, A., Coit, J., Levitt, K., Wu, S., Li, S., Kuo, J., Fan, K.: Buttercup: on network-based detection of polymorphic buffer overflow vulnerabilities. In: Proceedings of the Network Operations and Management Symposium (NOMS), pp. 235–248, (2004)
25. Pincus, J., Baker, B.: Beyond stack smashing: recent advances in exploiting buffer overflows. IEEE Security Privacy **2**(4), 20–27 (2004)
26. Kreibich, C., Crowcroft, J.: Honeycomb–creating intrusion detection signatures using honeypots. In: Proceedings of the Second Workshop on Hot Topics in Networks (HotNets-II), (2003)
27. Kolesnikov, O., Dagon, D., Lee, W.: Advanced polymorphic worms: evading IDS by blending in with normal traffic. In: College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, http://www.cc.gatech.edu/ ok/w/ok_pw.pdf, (2004)
28. Newsome, J., Karp, B., Song, D.: Paragraph: thwarting signature learning by training maliciously. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID), (2006)
29. Payer, U., Teufl, P., Lamberger, M.: Hybrid engine for polymorphic shellcode detection. In: Proceedings of the conference on detection of intrusions and malware and vulnerability assessment (DIMVA), pp. 19–31, (2005)
30. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM conference on Computer and communications security (CCS), pp. 290–299, (2003)
31. Aycock, J., deGraaf, R., Jacobson, M.: Anti-disassembly using cryptographic hash functions. Department of Computer Science, University of Calgary, Technical Report, pp. 793–824, (2005)
32. Venable, M., Chouchane, M.R., Karim, M.E., Lakhotia, A.: Analyzing memory accesses in obfuscated x86 executables. In: Proceedings of the conference on detection of intrusions and malware and vulnerability assessment (DIMVA), (2005)
33. Collberg, C.S., Thomborson, C.: Watermarking, tamper-proffing, and obfuscation: tools for software protection. IEEE Trans. Softw. Eng. **28**(8), 735–746 (2002)
34. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. University of Virginia, Technical Report CS-2000–12, (2000)
35. Madou, M., Anckaert, B., Moseley, P., Debray, S., Sutter, B.D., Bosschere, K.D.: Software protection through dynamic code mutation. In: Proceedings of the 6th International Workshop on Information Security Applications (WISA), pp. 194–206, (2005)
36. Schwarz, B., Debray, S., Andrews, G.: Disassembly of executable code revisited. In: Proceedings of the ninth working conference on reverse engineering (WCRE), (2002)
37. Prasad, M., cker Chiueh, T.: A binary rewriting defense against stack based overflow attacks. In: Proceedings of the USENIX annual technical conference, (2003)
38. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: Proceedings of the USENIX security symposium, pp. 255–270, (2004)
39. Cohen, F.B.: Operating system protection through program evolution. Comput. Sec. **12**(6), 565–584 (1993)
40. Metasploit project, http://www.metasploit.com/, (2006)
41. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Softw. Prac. Exp. **25**(7), 811–829 (1995)
42. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Proceedings of the International Conference on Compiler Construction (CC), (2004)
43. Noir, GetPC code (was: Shellcode from ASCII), http://www.securityfocus.com/ archive/82/327100/2006-01-03/1, June 2003
44. Ionescu, C.: GetPC code (was: Shellcode from ASCII), http://www.securityfocus.com/archive/82/327348/2006-01-03/1, July 2003
45. Wever, B.-J.: Alpha 2, (2004), http://www.edup.tudelft.nl/ bjwever/src/alpha2.c
46. Perriot, F., Ferrie, P., Ször, P.: Striking similarities. Virus Bull., pp. 4–6, (2002)

47. Obscou, Building IA32 ‘unicode-proof’ shellcodes. Phrack **11**(61), (2003)
48. Tubella, J., González, A.: Control speculation in multithreaded processors through dynamic loop detection. In: Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA), (1998)
49. McCanne, S., Leres, C., Jacobson, V.: Libpcap. http://www.tcpdump.org/, (2006)
50. Wojtczuk, R.: Libnids. http://libnids.sourceforge.net/, (2006)
51. jt, Libdasm. http://www.klake.org/~jt/misc/libdasm-1.4.tar.gz, (2006)
52. Apache Chunked Encoding Overflow. http://www.osvdb.org/838, (2002)
53. Microsoft Windows RPC DCOM Interface Overflow, http://www.osvdb.org/2100, (2003)
54. Microsoft Windows LSASS Remote Overflow, http://www.osvdb.org/5248, (2004)
55. Bell, J.R.: Threaded code. Comm. of the ACM. **16**(6), 370–372 (1973)
56. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 41–46, (2005)
57. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium, (2003)
58. Anagnostakis, K., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., Keromytis, A.D.: Detecting targeted attacks using shadow honeypots. In: Proceedings of the 14th USENIX Security Symposium, pp. 129–144, (2005)
59. Hsu, F.-H., Chiueh, T.-C.: CTCP: a transparent centralized tcp/ip architecture for network security. In: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC), pp. 335–344, (2004)
60. Liang, Z. Sekar, R.: Fast and automated generation of attack signatures: a basis for building self-protecting servers. In: Proceedings of the 12th ACM conference on Computer and communications security (CCS), pp. 213–222, (2005)
61. Dreger, H., Kreibich, C., Paxson, V., Sommer, R.: Enhancing the accuracy of network-based intrusion detection with host-based context. In: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), (2005)