Emulation-based Detection of Non-self-contained Polymorphic Shellcode

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] Institute for Infocomm Research, Singapore
kostas@i2r.a-star.edu.sg

Abstract. Network-level emulation has recently been proposed as a method for the accurate detection of previously unknown polymorphic code injection attacks. In this paper, we extend network-level emulation along two lines. First, we present an improved execution behavior heuristic that enables the detection of a certain class of non-self-contained polymorphic shellcodes that are currently missed by existing emulation-based approaches. Second, we present two generic algorithmic optimizations that improve the runtime performance of the detector. We have implemented a prototype of the proposed technique and evaluated it using off-the-shelf non-self-contained polymorphic shellcode engines and benign data. The detector achieves a modest processing throughput, which however is enough for decent runtime performance on actual deployments, while it has not produced any false positives. Finally, we report attack activity statistics from a seven-month deployment of our prototype in a production network, which demonstrate the effectiveness and practicality of our approach.

1 Introduction

Along with the phenomenal growth of the Internet, the number of attacks against Internet-connected systems continues to grow at alarming rates. From "one hostile action a week" 15 years ago [7], Internet hosts today confront millions of intrusion attempts every day [34]. Besides the constantly increasing number of security incidents, we are also witnessing a steady increase in attack sophistication. During the last few years, there has been a decline in the number of massive easy-to-spot global epidemics, and a shift towards more targeted and evasive attacks.

For example, attackers have been increasingly using techniques like polymorphism and metamorphism [28] to evade network-level detectors. Using polymorphism, the code in the attack vector —which is usually referred to as *shellcode*— is mutated so that each instance of the same attack acquires a unique byte pattern, thereby making fingerprinting of the whole breed very difficult. In its most naive form, the shellcode is encrypted using a simple algorithm, such as XOR-ing blocks of the original shellcode —which is also known as the *payload*— with a random key, and is prepended with a decryption routine that on runtime unveils and executes the encrypted payload.

Nowadays, the large and diverse number of polymorphic shellcode engines [1, 4, 9, 11, 13, 20, 23, 27, 33], along with their increased sophistication, makes imperative the need for effective and robust detection mechanisms. Along with the several research efforts towards this goal, we have recently proposed network-level emulation [22], a passive network monitoring approach for the detection of previously unknown polymorphic shellcode, which is based on the actual execution of network data on a CPU emulator. The principle behind network-level emulation is that the machine code interpretation of arbitrary data results to random code, which, when it is attempted to run on an actual CPU, usually crashes soon, e.g., due to the execution of an illegal instruction. In contrast, if some network request actually contains a polymorphic shellcode, then the shellcode runs normally, exhibiting a certain detectable behavior.

Network-level emulation does not rely on any exploit or vulnerability specific signatures, which allows the detection of previously unknown attacks. Instead, network-level emulation uses a generic heuristic that matches the runtime behavior of polymorphic shellcode. At the same time, the actual execution of the attack code on a CPU emulator makes the detector robust to evasion techniques such as highly obfuscated or self-modifying code. Furthermore, each input is inspected autonomously, which makes the approach effective against targeted attacks.

In this paper, we extend network-level emulation with an improved behavioral heuristic that allows the detection of a new class of polymorphic shellcodes, which are currently missed by the existing approach. The existing network-level emulation technique can detect only self-contained shellcode, which does not make any assumptions about the state of the vulnerable process. In this work, we enable the detection of a certain class of *non-self-contained* polymorphic shellcodes, which take advantage of a certain register that happens to hold the base address of the injected shellcode upon hijacking the instruction pointer. We also present two generic algorithmic optimizations that improve the runtime performance of the detector, and can be applied to network-level emulation irrespectively of the behavioral heuristic used. Finally, we report attack statistics from a real-world deployment of our prototype implementation, which we believe demonstrate the effectiveness and practicality of network-level emulation.

2 Related Work

The constant increase in the amount and sophistication of remote binary code injection attacks, and the consequent increase in the deployment and accuracy of defenses, have led to a coevolution of attack detection methods and evasion techniques.

Early approaches to network-level detection of zero-day worms relied on the identification of common byte sequences that are prevalent among multiple worm instances for the automated generation of NIDS signatures [14, 24]. Such approaches are effective only for fast spreading worms that do not use any form of payload obfuscation. As more tools for shellcode encryption and polymorphism became publicly available [1, 4, 9, 11, 13, 20, 23, 27, 33], subsequent automated signature generation approaches [16, 18] focused on the detection of polymorphic worms by identifying multiple common invariants among different worm instances. However, the first-level classifier on which such methods rely can result to evasion attacks [19].

An inherent limitation of the above approaches is that they are effective only after several instances of the same worm have reached the detector, which makes them

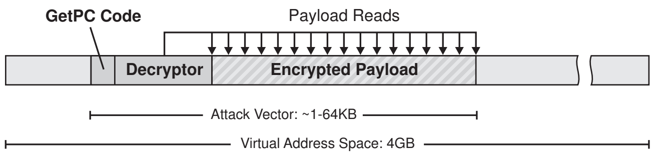

Fig. 1. A typical execution of a polymorphic shellcode using network-level emulation.

ineffective against targeted attacks. Content-based anomaly detection can also identify worms that employ a certain degree of polymorphism by alerting on traffic with anomalous content distributions [30, 31], although it is prone to blending attacks [12].

In face of extensive polymorphism, slow propagating worms, and targeted attacks, several research efforts turned to static binary code analysis on network traffic for identifying the presence of polymorphic shellcode. Initial approaches focused on the identification of the sled component that often precedes the shellcode [2, 29]. Recent works aim to detect the polymorphic shellcode itself using various approaches, such as the identification of structural similarities among different worm instances [15], control and data flow analysis [8, 32], or neural networks [21].

Static analysis, however, cannot effectively handle code that employs advanced obfuscation methods, such as indirect jumps and self-modifications, so carefully crafted polymorphic shellcode can evade detection methods based on static analysis. Dynamic code analysis using network-level emulation [22] is not hindered by such obfuscations, and thus can detect even extensively obfuscated shellcodes but is currently able to detect only self-contained polymorphic shellcode. Zhang et al. [35] propose to combine network-level emulation with static and data flow analysis for improving runtime detection performance. However, the proposed method requires the presence of a decryption loop in the shellcode, and thus will miss any polymorphic shellcodes that use unrolled loops or linear code, such as those presented in Sec. 3.

2.1 Network-level Emulation Overview

We briefly describe some aspects of the network-level emulation detection technique. The interested reader is referred to our previous work [22] for a thorough description of the approach and its implementation details.

The detector inspects the client-initiated data of each network flow, which may contain malicious requests towards vulnerable services. Any server-initiated data, such as the content served by a web server, are ignored. For TCP packets, the application-level stream is reconstructed using TCP stream reassembly. In case of large client-initiated streams, e.g., due to file uploads, only the first 64KB of the stream are inspected. Each input is mapped to a random memory location in the virtual address space of the emulator, as shown in Fig. 1. Since the exact location of the shellcode in the input stream is not known in advance, the emulator repeats the execution multiple times, starting from each and every position of the stream. We refer to complete executions from different positions of the input stream as *execution chains*. Before the beginning of a new execution, the state of the CPU is randomized, while any accidental memory modifications in the addresses where the attack vector has been mapped to are rolled back after the end of each execution. Since the execution of random code sometimes may not stop soon, e.g., due to the accidental formation of loop structures that may execute for a very large number of iterations, if the number of executed instructions in some execution chain reaches a certain *execution threshold*, then the execution is terminated.

The execution of polymorphic shellcode is identified by two key behavioral characteristics: the execution of some form of GetPC code, and the occurrence of several read operations from the memory addresses of the input stream itself, as illustrated in Fig 1. The GetPC code is used to find the absolute address of the injected code, which is mandatory for subsequently decrypting the encrypted payload, and involves the execution of some instruction from the `call` or `fstenv` instruction groups.

3 Non-self-contained Polymorphic Shellcode

The execution behavior of the most widely used type of polymorphic shellcode involves some indispensable operations, which enable network-level emulation to accurately identify it. Some kind of GetPC code is necessary for finding the absolute memory address of the injected code, and, during the decryption process, the memory locations where the encrypted payload resides will necessarily be read. However, recent advances in shellcode development have demonstrated that in certain cases, it is possible to construct a polymorphic shellcode which i) does not rely on any form of GetPC code, and ii) does not read its own memory addresses during the decryption process. A shellcode that uses either or both of these features will thus evade current network-level emulation approaches [22, 35]. In the following, we describe examples of both cases.

3.1 Absence of GetPC Code

The primary operation of polymorphic shellcode is to find the absolute memory address of its own decryptor code. This is mandatory for subsequently referencing the encrypted payload, since memory accesses in the IA-32 architecture can be made only by specifying an absolute memory address in a source or destination operand (except instructions like `pop`, `call`, or `fstenv`, which implicitly read or modify the stack). Although the IA-64 architecture supports an addressing mode whereby an operand can refer to a memory address relatively to the instruction pointer, such a functionality is not available in the IA-32 architecture.

The most common way of finding the absolute address of the injected shellcode is through the use of some form of GetPC code [22]. However, there exist certain exploitation cases in which none of the available GetPC codes can be used, due to restrictions in the byte values that can be used in the attack vector. For example, some vulnerabilities can be exploited only if the attack vector is composed of characters that fall into the ASCII range (or sometimes in even more limited groups such as printable-only characters), in order to avoid being modified by conversion functions like `toupper` or `isprint`. Since the opcodes of both `call` and `fstenv` have bytes that fall into these ranges, they cannot take part in the shellcode. In such cases, a possible workaround is to retrieve the address of the injected code through a register that during exploitation

```
 0  60000000 6A20          push 0x20           ; ecx points here
 1  60000002 6B3C240B      imul edi,[esp],0xb  ; edi = 0x160
 2  60000006 60            pusha               ; push all registers
 3  60000007 030C24        add ecx,[esp]       ; ecx = 0x60000160
 4  6000000a 6A11          push 0x11
 5  6000000c 030C24        add ecx,[esp]       ; ecx = 0x60000171
 6  6000000f 6A04          push 0x4            ; encrypted block size
 7  60000011 6826191413    push 0x13141926
 8  60000016 5F            pop edi             ; edi = 0x13141926
 9  60000017 0139          add [ecx],edi       ; [60000171] = "ABCD"
10  60000019 030C24        add ecx,[esp]       ; ecx = 0x60000175
11  6000001c 6817313F1E    push 0x1e3f3117
12  60000021 5F            pop edi             ; edi = 0x1E3F3117
13  60000022 0139          add [ecx],edi       ; [60000175] = "EFGH"
14  60000024 030C24        add ecx,[esp]       ; ecx = 0x60000179
    ...
```

Fig. 2. Execution trace of a shellcode produced by the "Avoid UTF8/tolower" encoder. When the first instruction is executed, `ecx` happens to point to address `0x60000000`.

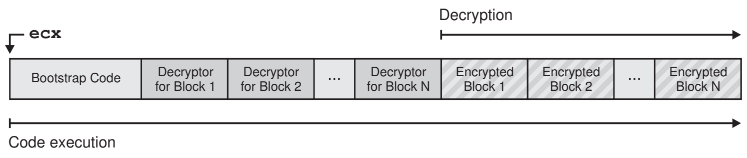

Fig. 3. Schematic representation of the decryption process for "Avoid UTF8/tolower" shellcode.

happens to point at the beginning of the buffer where the shellcode resides. If such a register exists, then the decoder can use it to calculate the address of the encrypted body.

Skape has recently published an alphanumeric shellcode engine that uses this technique [27]. Fig. 2 shows the execution trace of a shellcode generated using the implementation of the engine contained in Metasploit Framework v3.0 [1]. In this example, the register that is assumed to hold the base address of the shellcode is `ecx`. The shellcode has been mapped to address `0x60000000`, which corresponds to the beginning of the vulnerable buffer. When the control flow of the vulnerable process is diverted to the shellcode, the `ecx` register already happens to hold the value `0x60000000`. Instructions 0–5 calculate the starting address of the encrypted payload (`0x60000171`) based on its length and the absolute address contained in `ecx`.

The decryption process begins with instruction 7. An interesting characteristic of the decryptor is that it does not use any loop structure. Instead, separate transformation blocks comprising four instructions each (7–10, 11–14, ...) handle the decryption of different 4-byte blocks of the encrypted payload, as illustrated in Fig. 3. This results to a completely sequential flow of control for the whole decryption process. At the same time, however, the total size of the shellcode increases significantly, since for each four bytes of encrypted payload, an 11-byte transformation instruction block is needed.

3.2 Absence of Self-references

Another common characteristic of polymorphic shellcodes is that they carry the encrypted payload within the same attack vector, right after the decryptor code, as shown in Fig. 1. During execution, the decryptor necessarily makes several memory reads from

```
 0  bfff0000 54           push esp             ; esp points here
 1  bfff0001 58           pop eax              ; eax = BFFF0000
 2  bfff0002 2D6C2D2D2D   sub eax,0x2d2d2d6c   ; eax = 92D1D294
 3  bfff0007 2D7A555858   sub eax,0x5858557a   ; eax = 3A797D1A
 4  bfff000c 2D7A7A7A7A   sub eax,0x7a7a7a7a   ; eax = BFFF02A0
 5  bfff0011 50           push eax
 6  bfff0012 5C           pop esp              ; esp = BFFF02A0
 7  bfff0013 252D252123   and eax,0x2321252d   ; eax = 20012020
 8  bfff0018 2542424244   and eax,0x44424242   ; eax = 00000000
 9  bfff001d 2D2D2D2D2D   sub eax,0x2d2d2d2d   ; eax = D2D2D2D3
10  bfff0022 2D2D252D25   sub eax,0x252d252d   ; eax = ADA5ADA6
11  bfff0027 2D61675E65   sub eax,0x655e6761   ; eax = 48474645
12  bfff002c 50           push eax             ; [BFFF029C] = "EFGH"
13  bfff002d 2D2D2D2D2D   sub eax,0x2d2d2d2d   ; eax = 1B1A1918
14  bfff0032 2D5E5E5E5E   sub eax,0x5e5e5e5e   ; eax = BCBBBABA
15  bfff0037 2D79787878   sub eax,0x78787879   ; eax = 44434241
16  bfff003c 50           push eax             ; [BFFF0298] = "ABCD"
    ...
```

Fig. 4. Execution trace of a shellcode produced by the "Encode" engine. The shellcode is assumed to be placed on the stack, and `esp` initially points to the first instruction.

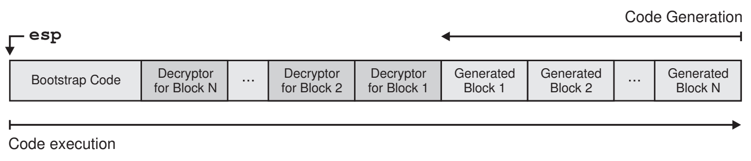

Fig. 5. Schematic representation of the decryption process for the "Encode" engine.

the addresses of the encrypted payload in order to decrypt it. These self-references can be used as a strong indication of the execution of polymorphic shellcode [22]. However, it is possible to construct a shellcode that, although it carries an encrypted payload, will not result to any memory reads from its own memory addresses.

Figure 4 shows the execution trace of a shellcode produced by an adapted version of the "Encode" shellcode engine [26], developed by Skape according to a previous description of Riley Eller [11]. In this case, the vulnerable buffer is assumed to be located on the stack, so `esp` happens to point to the beginning of the shellcode. Instructions 0–6 are used to set `esp` to point far ahead of the decryptor code (in higher memory addresses). Then, after zeroing `eax` (instructions 7–8), the decryption process begins, again using separate decryption blocks (9–12, 13–16, ...) for each four bytes of the encrypted payload. However, in this case, each decryption block consists only of arithmetic instructions with a register and an immediate operand, and ends with a `push` instruction. Each group of arithmetic instructions calculates the final value of the corresponding payload block, which is then pushed on the stack. In essence, the data of the encrypted payload are integrated into the immediate values of the arithmetic instructions, so no actual encrypted data exist in the initial attack vector.

Due to the nature of the stack, the decrypted payload is produced backwards, starting with its last four bytes. When the final decrypted block is pushed on the stack, the flow of control of the decryptor will "meet" the newly built payload, and the execution will continue normally, as depicted in Fig. 5. Notice that during the whole execution of the shellcode, only two memory reads are performed by the two `pop` instructions, but not from any of the addresses of the injected code.

4 Non-self-contained Polymorphic Shellcode Detection

4.1 Approach

Achieving the effective detection of a certain class of polymorphic shellcodes using network-level emulation requires the fulfillment of two basic requirements. First, the detector should be able to accurately reproduce the execution of the shellcode in exactly the same way as if it would run within the context of the vulnerable process. Second, it should be possible to identify a certain execution behavior pattern that can be used as a strict heuristic for the effective differentiation between the execution of polymorphic shellcode and random code. In this section, we discuss these two dimensions regarding the detection of non-self-contained shellcode.

Enabling Non-self-contained Shellcode Execution As discussed in the previous section, some shellcodes rely on a register that happens to contain the base address of the injected code, instead of using some form of GetPC code. Such shellcodes cannot be executed properly by the existing network-level emulation approach, since before each execution, all general purpose registers are set to random values. Thus, the register that is assumed to hold the base address will not have been set to the correct value, and the decryption process will fail. Therefore, our first aim is to create the necessary conditions that will allow the shellcode to execute correctly. In essence, this requires to set the register that is used by the shellcode for finding its base address to the proper value.

The emulator maps each new input stream to an arbitrary memory location in its virtual memory. Thus, it can know in advance the absolute address of the hypothetical buffer where the shellcode has been mapped, and as a corollary, the address of the starting position of each new execution chain. For a given position in the buffer that corresponds to the beginning of a non-self-contained shellcode, if the base register has been initialized to point to the address of that position, then the shellcode will execute correctly. Since we always know the base address of each execution chain, we can always set the base register to the proper value.

The problem is that it is not possible to know in advance which one of the eight general purpose registers will be used by the shellcode for getting a reference to its base address. For instance, it might be `ecx` or `esp`, as it was the case in the two examples of the previous section, or in fact any other register, depending on the exploit. To address this issue, we initialize all eight general purpose registers to hold the absolute address of the first instruction of each execution chain. Except the dependence on the base register, all other operations of the shellcode will not be affected from such a setting, since the rest of the code is self-contained. For instance, going back to the execution trace of Fig. 2, when the emulator begins executing the code starting with the instruction at address `0x60000000`, all registers will have been set to `0x60000000`. Thus, the calculations for setting `ecx` to point to the encrypted payload will proceed correctly, and the 9th instruction will indeed decrypt the first four bytes of the payload at address `0x60000171`. Note that the stack grows downwards, towards lower memory addresses, in the opposite direction of code execution, so setting `esp` to point to the beginning of the shellcode does not affect its correct execution, e.g. due to `push` instructions that write on the stack.

Behavioral Heuristic Having achieved the correct execution of non-self-contained shellcode on the network-level emulator, the next step is to identify a strict behavioral pattern that will be used as a heuristic for the accurate discrimination between malicious and benign network data. Such a heuristic should rely to as few assumptions about the structure of the shellcode as possible, in order to be resilient to evasion attacks, while at the same time should be specific enough so as to minimize the risk of false positives.

Considering the execution behavior of the shellcodes presented in the previous section, we can make the following observations. First, the absence of any form of GetPC code precludes the reliance on the presence of specific instructions as an indication of non-self contained shellcode execution, as was the case with the `call` or `fstenv` groups of instructions, which are a crucial part of the GetPC code. Indeed, all operations of both shellcodes could have been implemented in many different ways, using various combinations of instructions and operands, especially when considering exploits in which the use of a broader range of byte values is allowed in the attack vector. Second, we observe that the presence of reads from the memory locations of the input buffer during the decryption process is not mandatory, as demonstrated in Sec. 3.2, so this also cannot be used as an indication of non-self-contained shellcode execution.

However, it is still possible to identify some indispensable behavioral characteristics that are inherent to all such non-self-contained polymorphic shellcodes. An essential characteristic of polymorphic shellcodes in general is that during execution, they eventually unveil their initially concealed payload, and this can only be done by writing the decrypted payload to some memory area. Therefore, the execution of a polymorphic shellcode will unavoidably result to several memory writes to *different* memory locations. We refer to such write operations to different memory locations as "*unique writes*." Additionally, after the end of the decryption process, the flow of control will inevitably be transferred from the decryptor code to the newly revealed code. This means that the instruction pointer will move *at least once* from addresses of the input buffer that have not been altered before (the code of the decryptor), to addresses that have already been written during the same execution (the code of the decrypted payload). For the sake of brevity, we refer to instructions that correspond to code at any memory address that has been written during the same execution chain as "*wx-instructions*."

It is important to note that the decrypted payload may not be written in the same buffer where the attack vector resides [20]. Furthermore, one could construct a shellcode in which the unique writes due to the decryption process will be made to non-adjacent locations. Finally, wx-instructions may be interleaved with non-wx-instructions, e.g., due to self-modifications before the actual decryption, so the instruction pointer may switch several times between unmodified and modified memory locations.

Based on the above observations, we derive the following detection heuristic: *if at the end of an execution chain the emulator has performed W unique writes and has executed X wx-instructions, then the execution chain corresponds to a non-self-contained polymorphic shellcode*. The intuition behind this heuristic is that during the execution of random code, although there will probably be a lot of random write operations to arbitrary memory addresses, we speculate that the probability of the control flow to reach such a modified memory address during the same execution will be low. In the following, we elaborate on the details behind this heuristic.

Unique memory writes. The number of unique writes (W) in the heuristic serves just as a hint for the fact that at least a couple of memory locations have been modified during the same execution chain—a prerequisite for the existence of any wx-instructions. The parameter W cannot be considered as a qualitatively strong detection heuristic because the execution of random code sometimes exhibits a large number of accidental memory writes. The emulator does not have a view of the vulnerable process' memory layout, and thus cannot know which memory addresses are valid and writable, so it blindly accepts all write operations to any location, and keeps track of the written values in its own virtual memory. The decryption process of a polymorphic shellcode will too result to tens or even hundreds of memory writes. This makes the number of unique writes *per se* a weak indication for the execution of polymorphic shellcode, since random code sometimes results to a comparable number of writes.

Although this does not allow us to derive a threshold value for W that would be reached only during the execution of polymorphic shellcode, we can derive a lower bound for W, given that any regularly sized encrypted payload will require quite a few memory writes in order to be decrypted. Considering that the decryption of a 32-byte payload —a rather conservatively small size for a meaningful payload, as discussed in Sec. 5.2— would require at least 8 memory writes (using instructions with 4-byte operands), we set $W = 8$. This serves as a "negative" heuristic for deciding quickly the absence of shellcode, which effectively filters out a lot of execution chains with very few memory writes that cannot correspond to any functional polymorphic shellcode.

Execution of decrypted instructions. Although the number of unique writes alone cannot provide a strong positive indication for shellcode detection, we expected that the number of wx-instructions in random code would be very low, which would allow for deriving a definite detection threshold that would never be reached by random code. A prerequisite for the execution of code from a recently modified memory address is that the instruction pointer should first be changed to point to that memory address. Intuitively, the odds for this to happen in random code are quite low, given that most of the modified locations will be dispersed across the whole virtual address space of the emulator, due to the random nature of memory writes. Even if the control flow ever lands on such a memory address, most probably it will contain just a few valid instructions. In contrast, self-decrypting shellcode will result to the execution of tens or even hundreds of wx-instructions, due to the execution of the decrypted payload.

We conducted some preliminary experiments using real network traces and randomly generated data in order to explore the behavior of random code in terms of wx-instructions. The percentage of instruction chains with more than 8 unique writes and at least one wx-instruction was in the order of 0.01% for artificial binary data, while it was negligible for artificial ASCII data and real network traces. However, there were some rare cases of streams in which some execution chain contained as much as 60 wx-instructions. As we discuss in Sec. 5.2, the execution of the decrypted payload may involve less than 60 wx-instructions, so the range in which an accurate detection threshold value for X could exist is somehow blurred. Although one could consider the percentage of these outlying streams as marginal, and thus the false positive ratio as acceptable, it is still possible to derive a stricter detection heuristic that will allow for improved resilience to false positives.

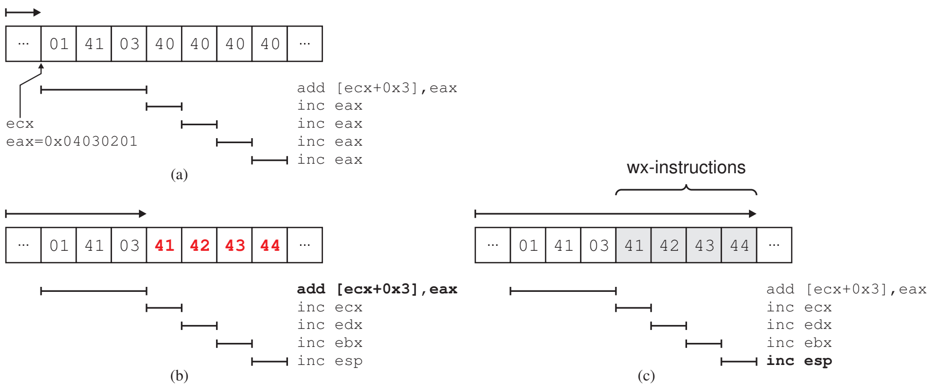

Fig. 6. An example of accidental occurrence of wx-instructions in random code.

Second-stage execution. The existence of some execution chains with a large number of wx-instructions in random code is directly related to the initialization of the general purpose registers before each new execution. Setting all registers to point to the address of the first instruction of the execution chain facilitates the accidental modification of the input stream itself, e.g., in memory addresses farther (in higher memory addresses) from the starting position of the execution chain. An example of this effect is presented in Fig. 6. Initially (Fig. 6a), when the flow of control reaches the instruction starting with byte `01`, `ecx` happens to point to the same instruction, and `eax` holds the value `0x04030201`. The effective address calculation in `add [ecx+0x3],eax` (Fig. 6b) involves `ecx`, and its execution results to a 4-byte memory write within the buffer, right after the `add` instruction. This simple self-modification causes the execution of four wx-instructions (Fig. 6c). Note that after the execution of these four wx-instructions, the flow of control will continue normally with the subsequent instructions in the buffer, so the same effect may occur multiple times.

In order to mitigate this effect, we introduce the concept of *second-stage execution*. For a given position in the input stream, if the execution chain that starts from this position results to more than 8 unique writes and has at least 14 wx-instructions,[3] then it is ignored, and the execution from this position is repeated eight times with eight different register initializations. Each time, only one of the eight general purpose registers is set to point to the starting location. The remaining seven registers are set to random values.

The rationale is that a non-self-contained shellcode that uses some register for finding its base address will run correctly both in the initial execution, when all registers point to the starting position, as well as in one of the eight subsequent second-stage executions—the one in which the particular base register being used by the decryptor will have been properly initialized. At the same time, if some random code enters second-stage execution, the chances for the accidental occurrence of a large number of wx-instructions in any of the eight new execution chains are significantly lower, since now only one of the eight registers happens to point within the input buffer.

[3] As discussed in Sec. 5.2, a functional payload results to at least 14 wx-instructions.

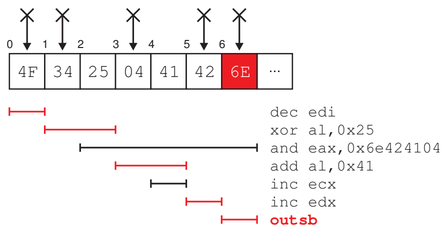

Fig. 7. Example of an illegal instruction path.

Although second-stage execution incurs an eight times increase in the emulation overhead, its is only triggered for a negligible fraction of execution chains, so it does not incur any noticeable runtime performance degradation. At the same time, it results to a much lower worst-case number of accidental wx-instructions in benign streams, as shown in Sec. 5.1, which allows for deriving a clear-cut threshold for X.

4.2 Performance Optimizations

Skipping Illegal Paths The main reason that network-level emulation is practically feasible and achieves a decent processing throughput is because, in the most common case, the execution of benign streams usually terminates early, after the execution of only a few instructions. Indeed, arbitrary data will result to random code that usually contains illegal opcodes or privileged instructions, which cannot take part in the execution of a functional shellcode. Although there exist only a handful of illegal opcodes in the IA-32 architecture, there exist 25 privileged instructions with one-byte opcodes, and several others with multi-byte opcodes. In the rest of this section, we use the term illegal instruction to refer to both privileged and actually illegal instructions.

A major cause of overhead in network-level emulation is that for each input stream, the emulator starts a new execution from each and every position in the stream. However, since the occurrence of illegal instructions is common in random code, there may be some instruction chains which all end to the same illegal instruction. After the execution of the first of these chains terminates (due to the illegal instruction), then any subsequent execution chains that share the same final instruction path with the first one will definitely end up to the same illegal instruction, if i) the path does not contain any control transfer instructions, ii) none of the instructions in the path was the result of a self-modification, and iii) the path does not contain any instruction with a memory destination operand. The last requirement is necessary in order to avoid potential self-modifications on the path that may alter its control flow. Thus, whenever the flow of control reaches any of the instructions in the path, the execution can stop immediately.

Consider for example the execution chain that starts at position 0 in the example of Fig. 7. Upon its termination, the emulator backtracks the instruction path and marks each instruction until any of the above requirements is violated, or the beginning of the input stream is reached. If any subsequent execution chain reaches a marked instruction, then the execution ceases immediately. Furthermore, the execution chains that would begin from positions 1, 3, 5, and 6, can now be skipped altogether.

Name	**Port Number**	**Number of streams**	**Total size**
HTTP	80	6511815	5.6 GB
NetBIOS	137–139	1392679	1.5 GB
Microsoft-ds	445	2585308	3.8 GB
FORTH-ICS	*all*	668754	821 MB

Table 1. Details of the client-initiated network traffic traces used in the experimental evaluation.

Kernel Memory Accesses The network-level detector does not have any information about the vulnerable process targeted by a particular attack. As already discussed, the emulator assumes that all accesses to any memory address are valid. In reality, only a small subset of these memory accesses would have succeeded, since the hypothetical vulnerable process would have mapped only a small subset of pages from the whole 4GB virtual memory space. Thus, memory writes outside the input buffer or the stack proceed normally and the emulator tracks the written values, while memory reads from previously unknown locations are executed without returning any meaningful data, since their contents are not available to the network-level detector. The execution cannot stop on such unknown memory references, since otherwise an attacker could hinder detection by interspersing instructions that read arbitrary data from memory locations known in advance to belong to the address space of the vulnerable process [22].

The network-level emulation approach assumes that the whole 4GB of virtual memory may be accessible by the shellcode. However, user-level processes cannot access the address space of the OS kernel. In Linux, the kernel address space begins at address `0xC0000000` and takes up the whole upper 1GB of the 4GB space. In Windows, the upper half of the 4GB space is allocated for kernel use. A functional shellcode would never try to access a memory address in the kernel address space, so any instructions in random code that accidentally try to access some kernel memory location can be considered illegal. For simplicity, the emulator assumes as legal all memory accesses up to `0xBFFFFFFF`, i.e., excludes only the common kernel space of both OSes, since it cannot know in advance which OS is being targeted.

5 Experimental Evaluation

5.1 Deriving a Robust Detection Threshold

The detection algorithm is based on a strict behavioral pattern that matches some execution characteristics of non-self-contained polymorphic shellcode. In order to be effective and practically applicable, a heuristic based on such a behavioral pattern should not falsely identify benign data as polymorphic shellcode. In this section, we explore the resilience of the detector to false positives using a large and diverse attack-free dataset.

We accumulated full payload packet traces of frequently attacked ports captured at FORTH-ICS and the University of Crete across several different periods. We also captured a two hour long trace of all the TCP traffic of the access link that connects FORTH-ICS to the Internet. Since we are interested in client-initiated traffic, which contains requests to network services, we keep only the packets that correspond to the

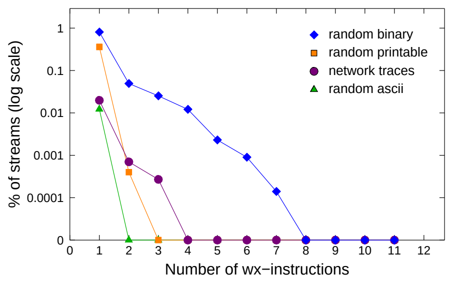

Fig. 8. Number of wx-instructions found in benign streams.

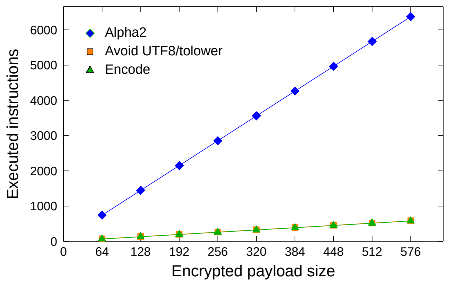

Fig. 9. Number of instructions required for complete decryption.

client-side stream of each TCP flow. For large flows, which for example may correspond to file uploads, we keep the packets of the first 64KB of the stream. Trace details are summarized in Table 1. Note that the initial size of the FORTH-ICS trace, before extracting the client-initiated only traffic, was 106GB. We also generated a large amount of artificial traces using three different kinds of uniformly distributed random content: binary data, ASCII-only data, and printable-only characters. For each type, we generated four million streams, totaling more than 160GB of data.

We tested our prototype implementation of the detection heuristic with second-stage execution enabled using the above dataset, and measured the maximum number of accidental wx-instructions among all execution chains of each stream. The execution threshold of the emulator was set to 65536 instructions. Figure 8 presents the results for the different types of random data, as well as for the real network streams (the category "network traces" refers collectively to all network traces listed in Table 1). We see that random binary data exhibit the largest number of wx-instructions, followed by printable data and real network traffic. From the four million random binary streams, 0.8072% contain an execution chain with one wx-instruction, while in the worst case, 0.00014% of the streams resulted to seven wx-instructions. In all cases, no streams were found to contain an execution chain with more than seven wx-instructions.

Based on the above results, we can derive a lower bound for the number of wx-instructions (parameter X of the detection heuristic) that should be found in an execution chain for flagging the corresponding code as malicious. Setting X=8 allows for no false positives in the above dataset. However, larger values are preferable since they are expected to provide even more improved resilience to false positives.

5.2 Non-self-contained Shellcode Detection

CPU execution threshold As discussed in Sec. 4.1, the execution of non-self-contained shellcode will exhibit several wx-instructions, due to the execution of the decrypted payload. However, a crucial observation is that most of these wx-instructions will occur *after* the end of the decryption process, except perhaps any self-modifications during the bootstrap phase of the decryptor [22, 33]. Thus, the emulator should execute the shellcode for long enough in order for the decryption to complete, and then for the decrypted payload to execute, for actually identifying the presence of wx-instructions. This means that the CPU execution threshold should be large enough to allow for the complete execution of the shellcode.

The number of executed instructions required for the complete decryption of the payload is directly related to i) the decryption approach and its implementation (e.g., decrypting one vs. four bytes at a time), and ii) the size of the encrypted payload. We used off-the-shelf polymorphic shellcode engines that produce non-self-contained shellcode to encrypt payloads of different sizes. We generated mutations of a hypothetical payload ranging in size from 64 to 576 bytes, in 64-byte increments, using the Avoid UTF8/tolower [1, 27], Encoder [11, 26], and Alpha2 [33] shellcode engines. The size of the largest IA-32 payload contained in the Metasploit Framework v3.0, `windows/adduser/reverse_http`, is 553 bytes, so we chose a slightly larger value of 576 bytes as a worst case scenario.

Figure 9 shows the number of executed instructions for the complete decryption of the payload, for different payload sizes. As expected, the number of instructions increases linearly with the payload size, since all engines spend an equal amount of instructions per encrypted byte during decryption. Alpha2 executes considerably more instructions compared to the other two engines, and in the worst case, for a 576-byte payload, takes 6374 instructions to complete. Thus, we should choose an execution threshold significantly larger than the 2048 instructions that is suggested in the existing network-level emulation approach [22].

Setting a threshold value for X A final dimension that we need to explore is the minimum number of wx-instructions (X) that should be expected during shellcode execution. As we have already mentioned, this number is directly related to the size of the encrypted payload: the smaller the size of the concealed code, the fewer the number of wx-instructions that will be executed. As shown in the previous section, the threshold value for X should be set to at least 8, in order to avoid potential false positives. Thus, if the execution of the decrypted payload would result to a comparable number of wx-instructions, then we would not be able to derive a robust detection threshold.

Fortunately, typical payloads found in remote exploits usually consist of much more than eight instructions. In order to verify the ability of our prototype implementation to execute the decrypted payload upon the end of the decryption process, we tested it with the IA-32 payloads available in Metasploit. Note that although the network-level emulator cannot correctly execute system calls or follow memory accesses to addresses of the vulnerable process, whenever such instructions are encountered, the execution continues normally (e.g., in case of an `int 80` instruction, the code continues as if the system call had returned). In the worst case, the `linux/x86/exec` family of payloads, which have the smallest size of 36 bytes, result to the execution of 14 instructions. All other payloads execute a larger number of instructions. Thus, based on the number of executed instructions of the smallest payload, we set X=14. This is a rather conservative value, given that in practice the vast majority of remote exploits in the wild are targeting Windows hosts, so in the common case the number of wx-instructions of the decrypted payload will be much higher.

Payloads targeting Linux hosts usually have a very small size due to the direct invocation of system calls through the `int 80` instruction. In contrast, payloads for Windows hosts usually involve a much higher number of instructions. Windows shellcode usually does not involve the direct use of system calls (although this is sometimes possible [5]), since their mapping often changes across different OS versions, and some crucial operations, e.g., the creation of a socket, are not readily offered through system calls. Instead, Windows shellcode usually relies on system API calls that offer a wide range of advanced functionality (e.g., the ability to download a file from a remote host through HTTP using just one call). This, however, requires to first locate the necessary library functions, which involves finding the base address of `kernel32.dll`, then resolving symbol addresses, and so on. All these operations result to the execution of a considerable number of instructions.

In any case, even a conservative value for X=14, which effectively detects both Linux and Windows shellcode, is larger enough than the seven accidental wx-instructions that were found in benign data, and thus allows for a strong heuristic with even more improved resilience to false positives.

5.3 Processing Throughput

In this section, we evaluate the raw processing throughput of the proposed detection algorithm. We have implemented the new detection heuristic on our existing prototype network-level detector [22], which is based on a custom IA-32 CPU emulator that uses interpretive emulation. We measured the user time required for processing the network traces presented in Table 1, and computed the processing throughput for different values of the CPU execution threshold. The detector was running on a PC equipped with a 2.53GHz Pentium 4 processor and 1GB RAM, running Debian Linux (kernel v2.6.18). Figure 10 presents the results for the four different network traces.

As expected, the processing throughput decreases as the CPU execution threshold increases, since more cycles are spent on streams with very long execution chains or seemingly endless loops. We measured that in the worst case, for port 445 traffic, 3.2% of the streams reach the CPU execution threshold due to some loop when using a threshold higher than 8192. This percentage remains almost the same even when using a threshold as high as 131072 instructions, which means that these loops would require a prohibitively large number of iterations until completion.

Overall, the runtime performance has been slightly improved compared to our previous network-level emulation prototype. Although the algorithmic optimizations presented in Sec. 4.2 offer considerable runtime performance improvements, any gain is compensated by the more heavy utilization of the virtual memory subsystem and the need to frequently undo accidental self-modifications in the input stream.

Port 80 traffic exhibits the worst performance among all traces, with an almost constant throughput that drops from 12 to 10 Mbit/s. The throughput is not affected by the CPU execution threshold because i) the zero-delimited chunk optimization[4] is not effective because HTTP traffic rarely contains any null bytes, and ii) the execution chains

[4] Given that in the vast majority of exploits the attack vector cannot contain a null byte, the detector skips any zero-byte delimited regions smaller than 50 bytes, since they are too small to contain a functional polymorphic shellcode [22].

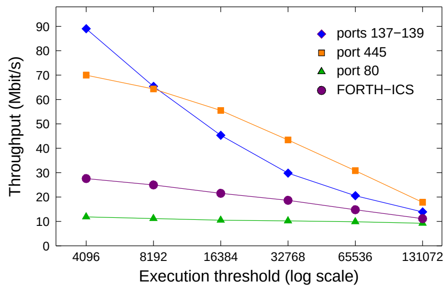

Fig. 10. Raw processing throughput for different execution thresholds.

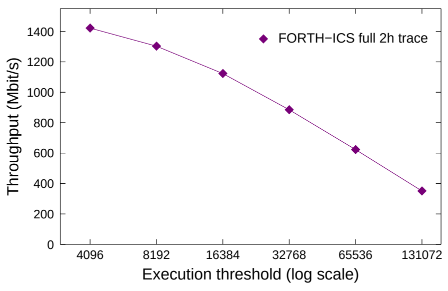

Fig. 11. Raw processing throughput for the complete 2-hour trace.

of port 80 traffic have a negligible amount of endless loops, so a higher CPU execution threshold does not result to the execution of more instructions due to extra loop iterations. However, ASCII data usually result to very long and dense execution chains with many one or two byte instructions, which consume a lot of CPU cycles.

We should stress that our home-grown CPU emulator is highly unoptimized, and the use of interpretive emulation results to orders of magnitude slowdown compared to native execution. It is expected that an optimized CPU emulator like QEMU [6] would boost performance, and we plan in our future work to proceed with such a change.

Nevertheless, the low processing throughput of the current implementation does not prevent it from being practically usable. In the contrary, since the vast majority of the traffic is server-initiated, the detector inspects only a small subset of the total traffic of the monitored link. For example, web requests are usually considerably smaller than the served content. Note that all client-initiated streams are inspected, in both directions. Furthermore, even in case of large client-initiated flows, e.g., due to file uploads, the detector inspects only the first 64KB of the client stream, so again the vast amount of the traffic will not be inspected. Indeed, as shown in Fig. 11, when processing the complete 106GB long trace captured at FORTH-ICS, the processing throughput is orders of magnitude higher. Thus, the detector can easily sustain the traffic rate of the monitored link, which for this 2-hour long trace was on average around 120 Mbit/s.

6 Real-world Deployment

In this section, we present some attack activity results from a real-world deployment of our prototype detector implementation. The detector is installed on a passive monitoring sensor that inspects the traffic of the access link that connects part of an educational network with hundreds of hosts to the Internet. The detector has been continuously operational since 7 November 2006, except a two-day downtime on January.

As of 14 June 2007, the detector has captured 21795 attacks targeting nine different ports. An overall view of the attack activity during these seven months is presented in Fig. 12. The upper part of the figure shows the attack activity according to the targeted port. From the 21795 attacks, 14956 (68.62%) were launched from 5747 external IP addresses (red dots), while the rest 6839 (31.38%) originated from 269 infected hosts in

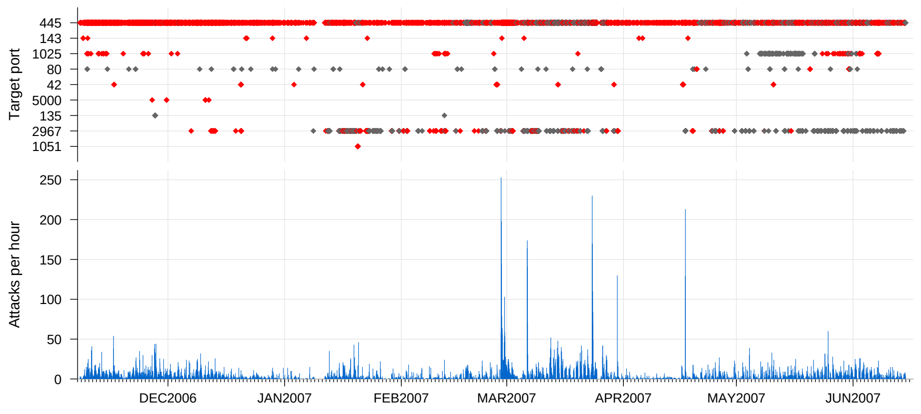

Fig. 12. Overall attack activity from a real-world deployment of our prototype detector.

the monitored network (gray dots). Almost one third of the internal attacks came from a single IP address, using the same exploit against port 445. The bottom part of the figure shows the number of attacks per hour of day. There are occasions with hundreds of attacks in one hour, mostly due to bursts from a single source that horizontally attacks all active hosts in local neighboring subnets. The vast majority of the attacks (88%) target port 445. Interestingly, however, there also exist attacks to less commonly attacked ports like 1025, 1051, and 5000. We should note that for all captured attacks the emulator was able to successfully decrypt the payload, while so far has zero false positives.

For each identified attack, our prototype detector generates i) an alert file with generic attack information and the execution trace of the shellcode, ii) a raw dump of the reassembled TCP stream, iii) a full payload trace of all attack traffic (both directions) in `libpcap` format,[5] and iv) the raw contents of the modified addresses in the virtual memory of the emulator, i.e., the decrypted shellcode.

Although we have not thoroughly analyzed all captured attacks, we can get a rough estimate on the diversity of the different exploitation tools, worms, or bots that launched these attacks, based on a simple analysis of the decrypted payloads of the captured polymorphic shellcodes. Computing the MD5 hash of the decrypted payload for all above attacks resulted to 1021 unique payloads. However, grouping further these 1021 payloads according to their size, resulted to 64 different payload size groups. By manually inspecting some of the shellcodes with same or similar lengths, but different MD5 hashes, we observed that in most cases the actual payload code was the same, but the seeding URL or IP address from where the “download and execute” shellcode would retrieve the actual malware was different. Our results are in accordance with previous studies [17] and clearly show that polymorphic shellcodes are extensively used in the wild, although in most cases they employ naive encryption methods, mostly for concealing restricted payload bytes.

[5] Anonymized full payload traces of some attacks are available from `http://lobster.ics.forth.gr/traces/`

7 Limitations

The increasing complexity of polymorphic shellcodes results to a corresponding increase in the processing time required for reasoning weather an input stream is malicious. Indeed, while self-contained polymorphic shellcode can effectively be detected using only 2K instructions per execution chain [22], non-self-contained shellcode, requires a CPU execution threshold in the order of 8K instructions. However, shellcode produced by advanced engines like TAPiON [4] sometimes requires up to 16K instructions for the complete decryption of an 128-byte payload [22], and can exceed 64K instructions for 512-byte payloads. Although such shellcodes use some form of GetPC code, and thus can be easily detected by the existing self-contained shellcode heuristic, if they begin to adopt non-self-contained techniques as those presented in this paper, then network-level emulation should be deployed with high execution thresholds, in the order of 128K instructions.

Fortunately, even in case we have to spend so many cycles per inspected input, network-level emulation is still practical, although with a reduced throughput, as we showed in Sec. 5.3. However, in the extreme case, an attacker could construct a decryptor that could spend millions of instructions, maybe even before the actual decryption process has begun at all, just for reaching the execution threshold before revealing any signs of polymorphic behavior [22]. Such "endless" loops are a well-known problem in the area of dynamic code analysis, and we are not aware of any effective solution so far. Fortunately, the percentage of benign streams that reach the execution threshold is under 3.2%, as discussed in Sec. 5.3, so if attackers start to employ such evasion techniques, network-level emulation can still be useful as a first-stage anomaly detector for application-aware NIDS like shadow honeypots [3], by considering as suspicious all streams that reach the execution threshold.

Finally, here we have considered only the class of non-self-contained shellcode that takes advantage of some register to get a reference to the absolute address of the injected code in order to decrypt. However, it could be possible to construct a shellcode that during decryption uses some data or code from memory locations with a priori known contents, which should remain constant across all vulnerable systems. Since the network-level detector lacks any host-level information, it would not be able to execute such shellcode properly. In general, however, the use of hard-coded addresses is avoided because it results in more fragile code [25], especially since address space randomization has become prevalent in popular OSes, and significantly complicates the implementation of polymorphic shellcode engines. In our future work, we plan to explore ways to augment the network-level detector with host-level context [10] for enabling the detection of a broader class of non-self-contained shellcodes.

8 Conclusion

In this paper, we have presented a novel approach for the detection of a certain class of non-self-contained polymorphic shellcodes using dynamic code analysis of network-level data. We have extended previous work on network-level emulation to correctly handle the execution and identify the behavior of polymorphic shellcodes that do not use any form of GetPC code, but instead rely on some register that happens during exploitation to contain the base address of the injected code. This demonstrates that in certain cases where some certain host-level state is used by the shellcode, detection at the network level is still possible.

Such advanced analysis comes at the cost of spending more CPU cycles per input, which reduces the runtime throughput of the detector, but still allows it to achieve a decent performance on real-world deployments. However, certain evasion methods are still possible, and the problem of effectively tackling them at the network-level remains open. Nevertheless, we believe that the ability to accurately detect previously unknown polymorphic shellcodes with virtually zero false positives, and the simplicity of its deployment, make network-level emulation an effective and practical defense method.

Acknowledgments This work was supported in part by the projects CyberScope and Miltiades, funded by the Greek General Secretariat for Research and Technology under contract numbers PENED 03ED440 and 05NON-EU-109, respectively, and by the FP6 project NoAH funded by the European Union under contract number 011923. M. Polychronakis and E. P. Markatos are also with the University of Crete. We deeply thank Periklis Akritidis and Elias Athanasopoulos for their valuable feedback on earlier versions of this work.

References

1. Metasploit project, 2006. `http://www.metasploit.com/`.
2. P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC)*, June 2005.
3. K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX Security Symposium*, pages 129–144, August 2005.
4. P. Bania. TAPiON, 2005. `http://pb.specialised.info/all/tapion/`.
5. P. Bania. Windows Syscall Shellcode, 2005. `http://www.securityfocus.com/infocus/1844`.
6. F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
7. S. M. Bellovin. There be dragons. In *Proceedings of the Third USENIX UNIX Security Symposium*, pages 1–16, 1992.
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. T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61), Aug. 2003.
10. 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.
11. R. Eller. Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms. `http://community.core-sdi.com/~juliano/bypass-msb.txt`.
12. P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee. Polymorphic blending attacks. In *Proceedings of the* 15^{th} *USENIX Security Symposium*, 2006.
13. K2. ADMmutate, 2001. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`.

14. H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 271–286, 2004.
15. 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.
16. Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 32–47, 2006.
17. J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *Proceedings of the 6th ACM SIGCOMM on Internet measurement (IMC)*, pages 53–64, 2006.
18. J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Security & Privacy Symposium*, pages 226–241, May 2005.
19. J. Newsome, B. Karp, and D. Song. Paragraph: Thwarting signature learning by training maliciously. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2006.
20. Obscou. Building IA32 'unicode-proof' shellcodes. *Phrack*, 11(61), Aug. 2003.
21. 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)*, pages 19–31, July 2005.
22. M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 54–73, July 2006.
23. Rix. Writing IA32 alphanumeric shellcodes. *Phrack*, 11(57), Aug. 2001.
24. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proc. of the 6th Symposium on Operating Systems Design & Implementation (OSDI)*, Dec. 2004.
25. sk. History and advances in windows shellcode. *Phrack*, 11(62), July 2004.
26. Skape. Shellcode text encoding utility for 7bit shellcode. `http://www.hick.org/code/skape/nologin/encode/encode.c`.
27. Skape. Implementing a custom x86 encoder. *Uninformed*, 5, Sept. 2006.
28. P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of the Virus Bulletin Conference*, pages 123–144, Sept. 2001.
29. T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proc. of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Oct. 2002.
30. K. Wang, G. Cretu, and S. J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the 8th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2005.
31. K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Proceedings of the 9th International Symposium on Recent Advanced in Intrusion Detection (RAID)*, 2006.
32. X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.
33. B.-J. Wever. Alpha 2, 2004. `http://www.edup.tudelft.nl/~bjwever/src/alpha2.c`.
34. V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: global characteristics and prevalence. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 138–147, 2003.
35. Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.