Hide and seek: worms digging at the Internet backbones and edges

Svetlana Gaivoronski
Computational Mathematics and Cybernetics dept.
Moscow State University
Moscow, Russia
Email: sadie@lvk.cs.msu.su

Dennis Gamayunov
Computational Mathematics and Cybernetics dept.
Moscow State University
Moscow, Russia
Email: gamajun@cs.msu.su

***Abstract*—The problem of malicious shellcode detection in high-speed network channels is a significant part of the more general problem of botnet propagation detection and filtering. Many of the modern botnets use remotely exploitable vulnerabilities in popular networking software for automatic propagation. We formulate the problem of shellcode detection in network flow in terms of formal theory of heuristics combination, where a set of detectors are used to recognize specific shellcode features and each of the detectors has its own characteristics of shellcode space coverage, false negative and false positive rates and computational complexity. Since the set of detectors and their quality is the key to the problem's solution, we will provide a survey of existing shellcode detection methods, including static, dynamic, abstract execution and hybrid, giving an estimation to the quality of the characteristics for each of the methods.**

***Keywords*-shellcode; malware; polymorphism; metamorphism; botnet detection;**

I. Introduction

Since the early 2000's and until the present time botnets are one of the key instruments used by cybercriminals for all kinds of malicious activity: stealing users' financial information, bank accounts credentials, organizing DDoS attacks, e-mail spam, malware hosting et cetera. Among the recent botnet activity we could mention the Torpig botnet, which was deeply investigated by the UCSB research group Torpig, the Zeus botnet involved in FBI' investigations which ended in arrest of over twenty people in September 2010 [6], and also the Kido/Conficker botnet, which has attracted the attention of security researchers since the end of 2008 and is still one of the most widespread trojan programs found on end users computers [4].

Despite of the fact that malware tends to propagate via web applications vulnerabilities, drive-by downloads, rogue AV software and infecting legitimate websites more often, the significance of remotely exploitable vulnerabilities in widespread networking software does not seem to have faded out in the following years, since the large installation base of the vulnerable program warrants very high infection rates in case of the zero-day attacks. Besides, drive-by downloads often make use of remotely exploitable vulnerabilities in the client software like Microsoft's Internet Explorer, Adobe Reader or Adobe Flash. A typical remotely exploitable vulnerability is a kind of memory corruption error - heap or stack overflows, access to the previously freed memory and other overflow vulnerabilities. Modern malware utilizes so called ”exploit packs”, commercially distributed suites of shellcodes for many different vulnerabilities, some of which may be unknown to the public. For example, the Conficker worm exploited several attack vectors for propagation: the MS08-67 vulnerability in Microsoft RPC service, dictionary attack for local NetBIOS shares and propagation via USB sticks autorun. Nevertheless, among all these propagation methods exploitation of the vulnerabilities in the networking software gives the attacker (or the worm) the best timing characteristics for botnet growth, because it requires no user interaction.

We could conventionally designate the following main stages of the botnets life cycle: propagation, privilege escalation on the infected computer, downloading trojan payload, linking to the botnet, executing commands from the botnet's C&C, removal from the botnet. Comparing the ease of botnet activity detection and differentiating it from normal Internet users activity, the propagation stage would be the most interesting as it involves computer attack, which is always an anomaly. The stages that follow successful infection - trojan extensions downloads, linking to botnet and receiving commands are usually made using ordinary application level protocols like HTTP or (rarely) IRC, different variations of P2P protocols, so that these communications are fairly easy to render to look like normal traffic. At the same time the propagation stage almost always involves shellcode transfer between attacker and victim, therefore it is easier to detect then other stages. This is why memory corruption attacks and their detection are important for modern Internet security.

A. *Shellcodes and memory corruption attacks*

A memory corruption error occurs when some code within the program writes more data to the memory, than the size of the previously allocated memory, or overwrites some internal data structures like malloc() memory chunks delimiters. One typical example of a memory corruption attack is stack overflow, where the attacker aims at overwriting the function return address with an address somewhere within

Activator	Decryption routine	Shellcode payload	Return address zone

Figure 1: Example of possible shellcode structure. Activator may be NOP-sled or GetPC code or alike.

the shellcode. Another example of a memory corruption attack is a heap overflow which exploits dynamic memory allocation/deallocation scheme in the operating system's standard library.

An example of a possible shellcode structure is shown at figure 1. Conditionally, we could break shellcodes into classes depending on which special regions they contain, where each shellcode region carries out some specific shellcode function, including detection evasion. For example, these could be regios of NOP-equivalent instructions (NOP-sled) or GetPC code as an activator, a decryption routine region for encrypted shellcodes, shellcode payload or return address zone.

In terms of classification theory we could define a shellcode as a set of continuous regions of executable instructions of the given architecture, where regions are associated by the control flow (following each other sequentially or linked to each other with control flow transfer instructions), and where one or more shellcode features are present simultaneously (i.e. it contains an activator, decryptor, shellcode payload zone or return address zone, associated by control flow).

There are significant numbers of existing and ongoing research activities which try to solve shellcode detection in network flow problem. These methods can be grouped into classes in two ways - by the type of analysis they perform (static, dynamic, abstract execution, hybrid) or by the types of shellcode features they are designed to detect (for example, activator, decryptor, shellcode payload, return address zone). An important observation is that most modern research papers are focused on IA32 (EM64-T) architecture, since most Internet-connected devices running Windows platform use this architecture and, besides, some of Intel Architecture instruction set features make memory corruption exploitation easier. This may change in the following decade when the broadband wireless connections for mobile devices become more common.

B. *Computation complexity problem*

Since we primarily aim at detecting network worms propagation (botnet growth) and not just remote exploitation of memory corruption vulnerabilities, our task has several certain peculiarities. Like any massive phenomenon worm propagation is best monitored in large scale, than at the end point of the attacked computer. This means that we should better try to detect worm propagation analyzing network data in transit at the Tier-2 channels or even Tier-1 channels. And in this case we inevitably fail because of the lack of computational power. There are two famous empiric laws which reflect the evolution of computation and computer networks - these are Moore's law and Gilder's law. Moore's law states that the processing power of a computer system available for the same price doubles every 18 months, and the Gilder's law says that the total bandwidth of communication systems triples every twelve months (see figure 2). The computational power of a typical computer system available for network channel analysis tends to grow slower than the throughput of the channel. The real-time restrictions for filtering devices also become more strict. For example, the worst case scenario for 1Gbps channel which is a flow of 64-byte IP packets at the maximum throughput gives us about 600ns average time for each packet analysis if we want to achieve wire-speed, and it gives only about 60ns in case of 10Gbps channel. This trend makes requirements for computational complexity of the algorithms utilized by network security devices more severe each year. That's why algorithms used for inline shellcode mitigation should have reasonable computational complexity and allow implementation in the custom hardware (FPGA, ASIC).

We also should not forget that backbone network channels like those connecting two or more different autonomous systems are especially sensitive to the false positives of the filtering device, because they lead to denial of service for the legitimate users.

In this paper we formulate the task of the malicious shellcode detection in the high-speed network channels as a multi-criteria optimization problem: how to build a shellcode classifier topology using a given set of simple shellcode feature classifiers, where each simple classifier is capable of detecting one or more simple shellcode features with zero false negative rates, given computational complexity and false positive rates within its shellcode classes, so that to provide the optimum aggregate false positive rates along with computational complexity. The key element of any solution of this task is the set of simple classifires. Thus, we provide a survey of the existing methods and algorithms of shellcode detection, which could be used as simple classifiers for the aggregate detector. In this survey we pay special attention to the class coverage, false positives rates and computational complexity of each method or algorithm. The structure of this paper is as follows. In the second section the classes of shellcode features are given and the main part of the section is the shellcode detection methods survey. In the third section we provide estimations of the key methods characteristics, which are essential for solving multi-criteria optimization problems. In the final two sections we discuss the results of the survey and suggest the formal task definition for building hybrid classifier as a oriented filtering graph of simple classifiers with optimal

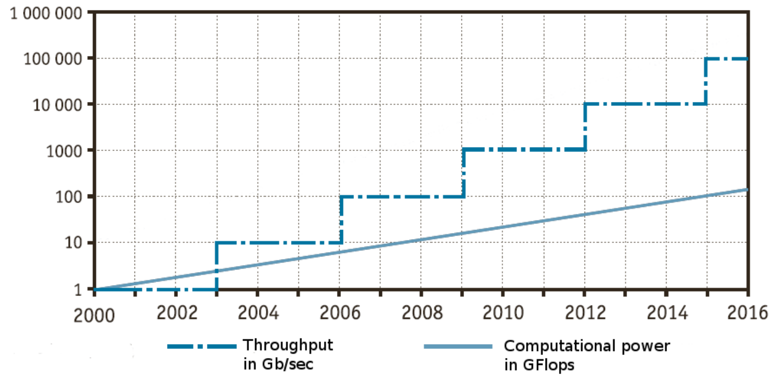

Figure 2: Moore and Gilder laws - the network channels throughput leaves the computational power behind.

computational complexity and false positive rates.

II. Shellcode detection methods

This section provides a classification of malicious objects and methods of shellcode detection. In addition, we will give a description of existing methods. For each method, we will briefly describe the basic idea. We will also describe classes of shellcode and their coverage, false positive rates and, where possible, we will give the computational complexity for the methods.

Let $S = \{Seq_1, \dots, Seq_r\}$ be a given set of sequences of executable instructions, later referred to as *object* S. We assume that all instructions in the object are valid instructions of the target processor. Let us define several definitions for S, using terminology from [1].

Let us consider a set of *features* $Mal = \{m_1, \dots, m_n\}$ of malicious instruction set (a malicious object) and a set of *features* $Leg = \{l_1, \dots, l_k\}$ of a legitimate set of instructions.

Suppose we are given a set M of *malicious* objects. Set M is covered by a finite number of subsets $K_1, \dots, K_l$:

$$M = \bigcup_{i=1}^{l} K_l.$$

Subset $K_j, j = \overline{1,l}$ is called *the class* of malware. Each class K_j is associated to the set of features $Mal(K_j)$ and $Leg(K_j)$ from the set of malicious features Mal and legitimate features Leg respectively. In addition, the partition of M to K_J classes conducted in a way that

$$Mal = \bigcup_{i=1}^{l} Mal(K_i)$$

and

$$Leg \neq \bigcup_{i=1}^{l} Leg(K_i)$$

in general.

Each class K_j is assigned with *elementary predicate*

$$P_j(S) = (S \in K_j),\ P_j(S) \in \{0, 1, \Delta\}$$

(object $S \in K_j$; $S \notin K_j$; unknown). The information about the occurrence of object in the class $K_1, \dots, K_l$ is encoded by vector $(\alpha_1 \alpha_2 \dots \alpha_l)$, $\alpha_i \in \{0, 1, \Delta\}$, $i = \overline{1,l}$.

Definition 1: Instruction set S is called a legitimate, if its information vector is null $|\tilde{\alpha}(S)| = 0$. In other words, the object is considered as legitimate iff it is not contained in any of the classes K_j of malicious set M.

Definition 2: Instruction set S is called malicious if the length of its information vector is equal to or greater than 1: $|\tilde{\alpha}(S)| \geq 1$. In other words, the object is considered as shellcode if it is contained at least in one of the classes K_j of malicious set M.

The problem of detecting malicious executable instructions is to calculate the values of the predicates $P_j(S) = (S \in K_j)$ and to construct information vector $\tilde{\alpha}^A(S)$, where A is the detection algorithm.

Definition 3: False negatives FN of algorithm A is the probability that the information vector of S resulted by algorithm A is null, but the veritable vector of object S is not null.

$$FN(A) = \mathbf{P}(|\tilde{\alpha}^A(S)| = 0 \mid |\tilde{\alpha}(S)| \geq 1)\ ,\ S \in M.$$

In other words, it is probability that a malicious object is not assigned to any of the classes K_j of malicious set M.

Definition 4: False positives FP of algorithm A is the probability that the length of information vector of object S returned by algorithm A is greater than or equal to 1, but the veritable vector of object S is null.

$$FP(A) = \mathbf{P}(|\tilde{\alpha}^A(S)| \geq 1 \mid |\tilde{\alpha}(S)| = 0)\ ,\ S \notin M.$$

In other words, it is the probability of classifying a legitimate object to at least one of the classes K_j of malicious set M.

A. *Shellcode features classification*

As previously mentioned, the entire set of malicious objects M is covered by the classes $K_1, \dots, K_l$: $M = \bigcup_{i=1}^{l} K_l$. Let us define the classes $K_1, \dots, K_l$ with respect to the structure of malicious code. Thus, the set M can be classified as follows:

1) Activators:

- K_{NOP_1} - class of objects containing simple NOP-sled - a sequence of nop (0x90) instructions ;
- K_{NOP_2} - objects containing one-byte NOP-equivalents sled;
- K_{NOP_3} - objects containing multi-byte NOP-equivalents sled;
- K_{NOP_4} - objects containing four-byte aligned sled;
- K_{NOP_5} - objects containing trampoline sled;
- K_{NOP_6} - objects containing obfuscated trampoline-sled;
- K_{NOP_7} - objects containing static analysis resistant sled;
- K_{GetPC} - objects containing GetPC code.

2) Decryptors:

- K_{SELF_UNP} - self-unpacking shellcode class;
- K_{SELF_CIPH} - self-deciphering shellcode class.

3) Payload:

- K_{SH} - non-obfuscated shellcode class;
- K_{DATA} - class of shellcode with data obfuscation. For example, ASCII character set can be replaced by UNICODE;
- K_{ALT_OP} - class of shellcode obfuscated by the insertion of alternative operators;
- K_R - class of shellcode, obfuscated by instruction reordering in the code;
- K_{ALT_I} - class of shellcode, obfuscated by replacing the instructions with instructions with the same operational semantics;
- K_{INJ} - class of shellcode, obfuscated by code injection;
- K_{MET} - class of metamorphic shellcode - shellcode whose body is changing with respect to semantic structure maintaining;
- K_{NSC} - (non-self-contained) - class of polymorphic shellcode which does not rely on any form of GetPC code, and does not read its own memory addresses during the decryption process.

4) Return address zone:

- K_{RET} - class of shellcode which can be detected by searching for the return address zone;
- K_{RET_+} - class of shellcode whose return address is obfuscated. For example, one can change the order of lower address bits. In this case, the control will be transferred to different positions of the stack, but always in any part of NOP-sled. In this case, the functionality of the exploit will not be compromised.

B. *Methods classification*

According to the principles at work, shellcode detection methods can be divided into the following classes:

- *static methods* - methods of code analysis without executing it;
- *abstract execution* - analysis of code modifications and accessibility of certain blocks of the code without a real execution. The analysis uses assumptions on the ranges of input data and variables that can affect the flow of execution;
- *dynamic methods* - methods that analyze the code during its execution;
- *hybrid methods* - methods that use a combination of static and dynamic analysis and the method of abstract interpretation.

From a theoretical point of view, static analysis can completely cover the entire code of the program and consider all possible objects S, generated from the input stream. In addition, static analysis is usually faster than dynamic. Nevertheless, it has several shortcomings:

- A large number of tasks which rely on the program's behavior and properties, can't be solved by using static analysis in general. In particular, the following theorems have been proved in the work of E. Filiol [13]:
  Theorem 1: Problem of detecting metamorphic shellcode by static analysis is undecidable.
  Theorem 2: The problem of detection of polymorphic shellcode is NP-complete in the general case.
- The attacker has the ability to create malicious code which is static analysis resistant. In particular, one can use various techniques of code obfuscation, indirect addressing, self-modifying code techniques, etc.

In contrast to static methods, dynamic methods are resistant to the code obfuscation and to the various anti-static analysis techniques (including self-modification). Nevertheless, the dynamic methods also have several shortcomings:

- they require much more overheads than static analysis methods. In particular, a sufficiently long chain of instructions can be required to conclude whether the program has malicious behavior or not;
- the coverage of the program is not complete: the dynamic methods consider only a few possible variants of program execution. Moreover, many significant variants of program execution can not be detected;
- the environment emulation in which the program exhibits its malicious behavior is difficult;
- there are detection techniques for program execution in a virtual environment. In this case, the program has the ability to change its behavior in order not to exhibit the malicious properties.

C. *Static methods*

A traditional approach for static network-based intrusion detection is signature matching, where the signature is a set of strings or regular expression. Signature based designs compare their input to known, hostile scenarios. They have the significant drawback of failing to detect variations of known attacks or entirely new intrusions. Signatures themselves can be divided into two categories: context-dependent and signatures that verify the behavior of the program.

One example of signature-based methods is **Buttercup** [12] - a static method that focuses on the search of the return address zone. The algorithm solution is simply to identify the ranges of the possible return memory addresses for existing buffer-overflow vulnerabilities and to check the values that lie in the fixed range of addresses. The algorithm considers the input stream, divided into blocks of 32 bits. The value of each byte in the block is compared with the ranges of addresses from the signatures. If the byte value falls into one of the intervals, an object S is considered as malicious. Formally,

$$|\tilde{\alpha}^{BUTTERCUP}(S)| \neq 0 \Leftrightarrow \exists I_j \in S \ : \ val(I_j) \in [LOWER, UPPER],$$

where $LOWER$ and $UPPER$ - lower and upper limits of the calculated interval, respectively. In the notions of introduced model, we assume that the second part of the expression is predicate $P_j(S)$, defining membership of an object S to one of the classes of malware. Since this method relies on known return addresses used in popular exploits, it becomes unusable when the target host utilizes address space layout randomization (ASLR). Static return addresses are rarely used in real-world exploits nowdays.

Another example of the signature-based methods is the **Hamsa** [14] - static method that constructs context-dependent signatures with respect to a malware training sample. The algorithm selects the set

$$\{S_i \mid \tilde{\alpha}_j(S_i) \neq \{0, \Delta\}\}$$

from the training information. Then the algorithm constructs a signature $Sig_j = \{T_1, \dots, T_k\}$ from that set. The signature itself is a set of tokens $T_j = \{I_{j_1}, \dots, I_{j_h}\}$, where I_{j_i} are instruction. In general, in [14] the following theorem is represented:

Theorem 3: The problem of constructing a signature Sig with respect to the parameter $\rho < 1$ such that $FP(Sig) \leq \rho$ is NP-hard.
The authors make the following assumptions in the problem: let the parameters $k^*, u(1), \dots, u(k^*)$ characterize a signature. Then, the token t added to the signature during signature generation iff

$$FP(Sig \bigcup \{t\}) \leq u(i).$$

When the signature is generated, the algorithm checks whether it matches to object S or not:

$$|\tilde{\alpha}^{HAMSA}(S)| \neq 0 \Leftrightarrow Sig_j \in S.$$

Another considered static signature-based method is **Polygraph** [15]. The approach builds context-dependent signatures. The algorithm takes different versions of the same object S' as a training set of objects $S_1, \dots, S_m$ for training information. Versions of S are generated by applying the operation of polymorphic changes for m times. With respect to learning information the Polygraph builds three types of signatures. If any of these signatures matches object S then S is considered as malware. The types of signatures are following: i) conjunction signatures Sig_C (consist of a set of tokens, and match a payload if all tokens in the set are found in it, in any order); ii) token-subsequence signatures Sig_{SUB} (consist of an ordered set of tokens); iii) Bayes signatures $Sig_B = \{(T_{B_1}, M_1), \dots, (T_{B_r}, M_r)\}$ (consist of a set of tokens, each of which is associated with a score, and an overall threshold). We define the following predicates:

$$P_C(S) = (Sig \in S) \text{ -}$$

predicate checks whether objects S matches to conjuction signature;

$$P_{SUB}(S) = (\forall i, j, m, n, k, t \ : \ T_{SUB_i} = \{I_m, \dots, I_n\}, T_{SUB_j} = \{I_k, \dots, I_t\} : i < j \Rightarrow m < n) \text{ -}$$

predicate checks if set of tokens in the objects is ordered;

$$P_B(S) = (\forall i : |T_{B_i}| \geq M_i) \text{ -}$$

predicate checks whether token exceeds threshold. Then the algorithm can be formally described as:

$$|\tilde{\alpha}^{POLYGRAPH^C}(S)| \neq 0 \Leftrightarrow P_C(S),$$
$$|\tilde{\alpha}^{POLYGRAPH^{SUB}}(S)| \neq 0 \Leftrightarrow P_C(S) \& P_{SUB}(S),$$
$$|\tilde{\alpha}^{POLYGRAPH^B}(S)| \neq 0 \Leftrightarrow P_B(S).$$

Among the methods of static analysis, which generating the signature of program behavior, we have considered the method of **structural analysis** [9]. Let us call it Structural in the rest of paper. By training on a sample of malicious objects $S_1, \dots, S_m$ the approach constructs a signature base of program behavior. The object S is considered as malware if it matches any signature in the base. The method checks whether object matches a signature contained in the base by following steps:

- program structure is identified by analyzing the control flow graph (CFG);
- program objects are identified by CFG coloring technique;
- for each signature and for each built program structure the approach analyses, whether they are polymorphic modifications of each other.

Nevertheless, a simple comparison of the control flow graphs is ineffective due to the fact that this isn't robust to the simplest modifications. The authors of the method propose the following modification: subgraphs containing

k vertices are identified. Identification of the subgraph is carried out as follows:

- first, the adjacency matrix is built. The adjacency matrix of a graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (v_i, v_j) according to whether there is an edge from v_i to v_j or not;
- second, a single fingerprint is produced by concatenating the rows of the matrix;
- additionally, the calculation of fingerprints extended to account for colors of verticles (graph is colored according to the type of verticle). This is done by first appending the (numerical representation of the) color of a node to its corresponding row in the adjacency matrix.

Definition 5: Two subgraphs are related if they are isomorphic and their corresponding vertices are colored the same.

Definition 6: Two control flow graphs are related if they contain related K-subgraphs (subgraph containing k vertices). It is believed that if $CFG(S)$ are related to any control flow graph of a malicious object, then the object S itself is malicious.

Let $\{ID\}$ be the set of subgraphs identifiers. Subgraphs contained in the malicious objects from the training data. We define the predicate

$$P_{ST} = id(S) \in \{ID\}.$$

Thus,

$$|\tilde{\alpha}^{Structural}(S)| \neq 0 \Leftrightarrow P_{ST}(S).$$

The **Stride** [10] algorithm is NOP-sled detection method. STRIDE is given some input data, such as a URL, and searches each and every position of the data to find a sled. STRIDE can be formally described as follows: it forms object S from the input stream by disassembling, starting at offset $i+j$ of the input data, for all $j \in \{0, \dots, n-1\}$. It is believed that the input stream contains a NOP-sled of length n:

$$|\tilde{\alpha}^{STRIDE(n)}(S)| \neq 0,$$

if the object $S = \{I_1, \dots, I_k\}$ satisfies the following conditions:

- $k \geq n$;
- $\exists i \ : \ \forall j \ : \ j = i, \dots, i+n \Rightarrow (I_j \neq Privileged) \ || \ (\exists k : i \leq k < j \ I_k = JMP)$

In other words, it is believed that a sled of length n starts at position i if it is reliably disassembled from each and every offset $i+j$, $j \in \{0, \dots, n-1\}$ (or from each of the 4th byte) and in any subsequence of S privileged instruction isn't met (or a jump instruction is encountered along the way).

There is an algorithm **Racewalk** [11] which improves performance of the algorithm STRIDE through the decoded instructions caching. Moreover, Racewalk uses pruning techniques of instructions that are not a valid NOP-sled (for example, if we meet an invalid or a privileged instruction at some position h, it is obvious that the run from offset $j = h\%4$ is invalid. Consequently, the object S formed from the offset $j = h\%4$ will not appear in any of the classes $K_{NOP_1}, \dots, K_{NOP_7}$). Racewalk also uses the instruction prefix tree construction to optimize the process of disassembling.

Styx [8] is a static analysis method, based on CFG analyzing. Object S is believed to be malware if sliced CFG contains cycles. Cycles in the sliced CFG indicate the polimorphic behavior of the object. For such an object the signature is generated in order to use it in its signature base.

Given the object S algorithm builds a control flow graph (CFG). The vertices are the blocks of instruction chains. Such blocks do not contain any transitions. The edges are the corresponding transitions between the blocks. All the blocks in the graph can be divided into three classes:

- valid (the branch instruction at the end of the block has a valid branch target);
- invalid (the branch target is invalid);
- unknown (the branch target is unknown).

Styx constructs a sliced GFG from control flow graph. All invalid blocks and blocks to which invalid ones have the transitions are removed from sliced CFG. Some of the blocks are excluded as well using the technique of data flow analysis, described in [16]. From sliced CFG Styx constructs a set of all possible execution chains of instructions. Next, method considers each of chains to check whether it contains cycles or not. To formalize the algorithm we describe the following predicates. Let $SIG = \{Sig_1, \dots, Sig_n\}$ be the signatures base which constructed from training information. Thus,

$$P_{SIG}(S) = \exists i \ : \ (Sig_i \in SIG) \ \& \ Sig_i \subset S$$

is the predicate which verifies that the object matches to one of the previously generated signatures. $P_{cycle}(S)$ is the predicate which checks sliced CFG of S for cycles. Consequently,

$$|\tilde{\alpha}^{STYX}(S)| \neq 0 \Leftrightarrow P_{SIG}(S) \ || \ P_{cycle}(S).$$

In contrast to this algorithm, a method **SigFree** [17] staticaly analyses not CFG but an instruction flow graph (IFG). Vertices of CFG contain blocks of instruction while IFG vertices contain instructions only. An object is considered as malware if its behavior conforms to the behavior of real programs, rather than a random set of instructions. Such heuristic restricts applicability of method on the channels such that the profile of the traffic allows for the transfer of executable programs.

Definition 7: An instruction flow graph (IFG) is a directed graph $G = (V, E)$ where each node $v \in V$ corresponds to an instruction and each edge $e = (v_i, vj) \in E$

corresponds to a possible transfer of control from instruction v_i to instruction v_j .

The analysis is based on the assumption that a legitimate object S, consisting of instructions encountered in the input stream, can not be a fragment of a real program. Real programs are assigned with two important properties:

1) The program has specific characteristics that are induced by the operating system on which it is running, for example calls to the operating system or kernel library. A random instruction sequence does not carry this kind of characteristics.
2) The program has a number of useful instructions which affects the results of the execution path.

With respect to these properties, the method provides two schemes of IFG analysis. In the first scheme SigFree based on training information constructs a set $\{TEMPL\}$ of instructions call templates. Then algorithm checks whether object S satisfies these patterns or not. Let us describe the predicate

$$P_1 = \exists t \in \{TEMPL\} \ : \ t \in IFG(S)$$

which checks if IFG of S satisfies to any of the templates. Thus,

$$|\tilde{\alpha}^{SigFree_1}(S)| \neq 0 \Leftrightarrow P_1(S).$$

The second scheme is based on an analysis of the data stream. In this scheme, each variable can be mapped from the set

$$Q = \{U, D, R, DD, UR, DU\},$$

where the six possible states of the variables are defined as following. State U : undefined; state D: defined but not referenced; state R: defined and referenced; state DD: abnormal state define-define; state UR: abnormal state undefine-reference; and state DU : abnormal state define-undefine. SigFree constructs for an object S state variables diagram - an automaton

$$DSV = (Q, \Sigma, \delta, q_0, F),$$

where Σ is the alphabet, consisting of instruction of object S, and $q_0 = U$ is the initial state. If there is the transition to the final (abnormal state) when parsing S, it is believed that the instruction is useless. All useless instructions are excluded from the object S, resulting in an object $S' \subset S$. Let us describe the following predicate:

$$P_2(S) = |S'| > K,$$

where K - threshold. Thus,

$$|\tilde{\alpha}^{SigFree_2}(S)| \neq 0 \Leftrightarrow P_2(S).$$

There is an algorithm **STILL** [18] which improves the method $SigFree$. The method based on techniques to detect self-modifying and indirect jump exploit code are called static taint analysis and initialization analysis. The method is based on the assumption that self-modifying code and code using the indirect jump, must obtain an absolute address of the exploit payload. With respect of this the method searhes subset $S' \in S$ which obtains the absolute address of the payload at runtime. The variable that records the absolute address is marked as *tainted*. The method uses the static taint analysis approach to track the tainted values and detect whether tainted data are used in the ways that could indicate the presence of self-modifying and indirect jump exploit code. The variable can infect others through data transfer instructions (`push, pop, move`) and instructions that perform arithmetic or bit-logic operations (`add, sub, xor`).

The method uses initialization analysis in order to reduce the false positive rates. The analysis is based on the assumption that the operands of self-modifying code and code using the indirect transitions, must be initialized. If not, object s is considered as legitimate. Formally, $P_1 = tainted(S)$, $P_2 = initialized(S)$,

$$|\tilde{\alpha}^{STILL}(S)| \neq 0 \Leftrightarrow P_1(S) \& \neg P_2(S).$$

Semantic-aware malware detection [19] is a signature-based approach. The method creates a set of behavior signature patterns by training on a sample of malicious objects. The object S is considered as malware if its behavior conforms to at least one pattern from this set.

In [19] authors have proved the following theorem:

Theorem 4: The problem of determining whether S satisfies a template T of a program behavior is undecidable.

Thus, the authors notice that their method can not have full coverage of classes of malicious programs. The method identifies a malicious object to a limited number of program modification techniques. The algorithm constructs a set $\{T\}$ of patterns of a programs malicious behavior. It is believed that the object S matches the pattern, if the following conditions are satisfied:

- The values in the addresses, which were modified during execution, are the same after the template execution with the appropriate context;
- A sequence of system calls in template is a subsequence of system calls in S;
- If the program counter at the end of executing the template T points to the memory area whose value changed, then the program counter after executing S should also point into the memory area whose value changed.

In order to check whether object S matches the behavior pattern, the method checks that the vertices of the template correspond to vertices of S. The method also implements the construction of"def-use"ways and its checking. *Matching of template nodes to program nodes* is carried out by constructing a control flow graph CFG, with respect to the following rules (we also describe the predicate $P_1(S)$ checking whether nodes match each other) :

- A variable in the template can be unified with any program expression, except for assignment expressions;
- A symbolic constant in the template can only be unified with constant in S;
- The function memory can be unified with the function memory only;
- An external function call in the template can only be unified with the same external function call in the program.

Preservation of def-use paths. A def-use path is a sequence of template nodes (or $CFG(S)$). The first node of def-use path defines the variable and the last uses it. Each def-use path in a template should correspond to the program def-use path. Next, method checks whether a variable is stored in an invariant meaning or not in the paths. To solve the problem of preservation of the variable using the following procedures are implemented:

- first, the NOP-sled lookup using simple signature matching;
- second, search of such code fragments in which values of variables are not preserved. If found, the corresponding fragment of code is executed with a random initial state;
- finally, using the theorem prover like the Simplify method [20] or the UCLID method [21].

We define the predicate P_2 which checks the Preservation of def-use paths. Thus, $T \sim S \Leftrightarrow P_1(S)\ \&\ P_2(S)$. The algorithm itself can be formaly described as following:

$$|\tilde{\alpha}^{Semantic\_aware}(S)| \neq 0 \Leftrightarrow \exists T_i \in \{T\}\ :\ T_i \sim S.$$

D. Dynamic methods

One example of the dynamic method is the emulation method (**Emulation**) proposed by Markatos et al in [23]. The main idea of the approach is to analyze the chain of instructions received during execution in a virtual environment. The execution starts from each and every position of the input buffer since the position of the shellcode is not known in advance. Thus, the method generates a set of objects

$$\{S'_i \mid S'_i \subset S\}$$

from object S. If at least one of the objects S_i satisfies the following heuristics, object S is considered as malware. These heuristics include the execution of some form of getPC code by an execution chain of S'_i; another heuristic is checking whether the number of the memory accesses excess a given threshold. The object S'_i is considered as legitimate if during its execution an incorrect or privileged instruction was met. Let us define the following predicates:

$$P_1(S_i) = getPC \in S_i\ \&\ mem\_access\_number(S_i) \geq Thr,$$

where Thr is threshhold;

$$P_2(S_i) = \forall j\ :\ I_j \in S_i\ \&\ invalid(I_j).$$

Thus, [23] can be formally described as:

$$|\tilde{\alpha}^{Emulation}(S)| \neq 0 \Leftrightarrow \exists i\ :\ S_i \subset S\ \&\ P_1(S_i)\ \&\ \neg P_2(S_i).$$

Method **NSC emulation** [26] is an extension of [23]. The method focuses on non-self-contained (NSC) shellcode detection. The execution of executable chains also starts from each and every position of the input buffer. Object S is considered as malware, if it satisfies the following heuristic. Let *unique writes* be the write operations to different memory locations and let *wx-instruction* be an instruction that corresponds to code at any memory address that has been written during the chain execution. Let W and X be thresholds for the unique writes and wx-instructions, respectively. The object belongs to the class K_{NSC}, if after its execution emulator has performed at least W unique writes ($P_1(S) = unique\_writes \geq W$) and has executed at least X wx-instructions ($P_2 = wx \geq X$). Thus,

$$|\tilde{\alpha}^{NSC}(S)| \neq 0 \Leftrightarrow P_1(S)\ \&\ P_2(S).$$

Another method, which uses emulation is **IGPSA** [25]. The information about instruction is processed by automaton. All the instructions are categorized into five categories, represented by patterns $P_1, \ldots, P_5$. If an instruction writes PC into certain memory location, it is categorized into P_1; if it reads PC from the memory, it belongs to P_2; if it reads from memory location the instruction sequence resides in, it belongs to P_3; if it writes data into memory location PC, it belongs to P_4; otherwise it belongs to P_5. Method generates a sequence of transformed patterns W which consists of elements of the set $\{P_1, \ldots, P_5\}$. Thus, the object classification problem is the problem of determining whether its transformed pattern sequence W is accepted by atomaton

$$IGPSA = (Q, \Sigma, \delta, q_0, F),$$

where Q is the set of states, $\Sigma = \{P_1, \ldots, P_5\}$ is the alphabet, $\delta\ :\ Q \times \Sigma \rightarrow Q$ is the transition function, q_0 is the initial state and F is set of final states. Each state corresponds to polymorphic shellcode behavior. Let us describe the predicate $P(S)$ which checks whether W is accepted by IGPSA. Formally,

$$|\tilde{\alpha}^{IGPSA}(S)| \neq 0 \Leftrightarrow P(S).$$

E. Hybrid methods

One of the examples of the hybrid method is the method for detecting self-decrypting shellcode [24], proposed by K. Zhang. Let us call it **HDD** in the rest of the paper. The static part of the method includes two-way traversal and backward data flow analysis. By which the analysis method finds seeding subsets of instructions of S. The presence of malicious behavior is verified by the emulation of these subsets.

Firstly, static analysis method performs recursive traversal analysis of the instruction flow, starting at the seeding instruction. A seeding instruction that can demonstrate the behavior of $GetPC$ code (for example, `call, fnstenv`, etc.). The method starts the backward analysis, if a target instruction, an instruction that is either (a) an instruction that writes to memory, or (b) a branching instruction with indirect addressing, is encountered during the forward traversal. The method follows backwards the def-use chain in order to determine the operands of the target instruction. Then the method checks such chains $S_i \subset \{two\_way\_analysis(S)\}$ for the presence of cycles ($P_1(S_i)$). Moreover, it checks whether chains write to memory in the code address space (that fact is considered as self-modification behaviour). Let it be the $P_2(S_i)$ predicate. Let us also consider

$$P_3(S_i) = \forall j \ : \ I_j \in S_i \ \& \ invalid(I_j).$$

Thus,

$$|\tilde{\alpha}^{Hybrid\_dec\_detection}(S)| \neq 0 \Leftrightarrow \exists i \ : \ S_i \subset \{two\_way\_analysis(S)\} \ \& \ P_1(S_i) \ \& \ P_2(S_i) \ \& \ \neg P_3(S_i).$$

Another hybrid method is **PolyUnpack** [27]. This method is based on statical constructing of a program model and verification of this model by the emulation technique. The object S is said to be legitimate if it does not produce any data to be executed. Otherwise, the object is a self-extracting program. At the stage of static analysis, the object S is divided into code blocks and data blocks. These code blocks, separated by blocks of data, are a sequence of instructions $Sec_0, \ldots, Sec_n$, which represent the program's model. The statically derived model and object S are then transited into the dynamic analysis component where S is executed in an isolated environment. The execution is paused after each instruction and its execution context is compared with the static code model. If the instruction corresponds to the static model, then the execution continues. Otherwise, the object S is considered as malware. Let us describe the predicate $P(S)$ which checks whether object S satisfies its static code model. Then we can formally describe PolyUnpack as

$$|\tilde{\alpha}^{PolyUnpack}(S)| \neq 0 \Leftrightarrow \neg P(S).$$

F. *Methods of abstract execution*

At the present day, this class represented the only method that is called **APE** [28]. APE is NOP-sled detection method, which is based on finding sufficiently long sequences of valid instructions, whose operands in memory are in the protected address space of the process. There are a small number of positions in the experimental data, from which abstract execution should be started, which are chosen in order to reduce the computational complexity. The abstract execution is used to check the instruction's correctness and validity.

Definition 8: A sequence of bytes is correct, if it represents a single valid processor instruction. A sequence of bytes is valid if it is correct and all memory operands of the instruction reference the memory addresses that the process which executes the operation is allowed to access.

The number of correct instructions, which are decoded from each selected position, are denoted as *MEL* (Maximum Executable Length). It is possible that a byte sequence contains several disjoint abstract execution flows and the MEL denotes the length of the longest. The NOP-sled is believed to be found in S, if the value of MEL reaches a certain threshold Thr. Formally,

$$|\tilde{\alpha}^{APE}(S)| \neq 0 \Leftrightarrow (MEL \geq Thr).$$

III. Evaluation and discussion

This section provides an analysis and comparison of the above methods with respect to three criteria: the completeness of classes handling, the false positive rates and the computational complexity. The key difficulty here is that all the research papers observed in this paper use completely different testing conditions and testing datasets. Therefore it is not very helpful to compare the published false positives rates or throughput of the algorithms. For example, the STRIDE method was tested using only HTTP URI dataset, where the possibility of finding executable byte sequences is indeed relatively low. Some of the methods like SigFree are designed to detect "meaningful" executables and distinguish them from random byte sequences which only look like executable, but such method would definitely have high false positive rates when used for shellcode detection in the network channel where ELF executable transfer is quite normal. This means that the results provided in the original research papers can not be used directly for solving the problem of aggregate classifier generation. A real performance and false positive profiling should be performed, with some kind of representative dataset and a solid experiment methodology.

But for the task of the survey and making some preliminary relative comparison of the detection methods within the same shellcode feature classes the data provided in the original research papers could be useful. Therefore, we collected it into a series of summary tables, along with short descriptions of the testing conditions. The computational complexity estimation was made using the algorithm descriptions from the research papers and general knowledge of computational complexity of the typical tasks like emulation or sandboxing. The drawback of such estimation is that it gives only classes of complexity, not the real throughput in any given conditions. The actual throughput of the considered methods was analytically evaluated for the normalized machine 2.53 GHz Pentium 4 processor and 1 GB RAM with running Linux on it. Throughput is considered below when discussing the advantages and disadvantages of the methods.

Table I shows the comparison results for the completeness of classes coverage.

	Buttercup	Hamsa	Polygraph	Stride	Racewalk	Styx	Structural analysis	SigFree	STILL	Semantic aware	Emulation	HDD	NSC	IGPSA	PolyUnpack	APE
K_{NOP_1}		▨	▨	▨	▨			▨	▨							▨
K_{NOP_2}		▨	▨	▨	▨			▨	▨							▨
K_{NOP_3}		▨	▨	▨	▨			▨	▨							▨
K_{NOP_4}		▨	▨	▨	▨			▨	▨							▨
K_{NOP_5}		▨	▨	▨	▨			▨	▨							▨
K_{NOP_6}		▨	▨	▨	▨			▨	▨							
K_{NOP_7}						▨		▨	▨		▨	▨			▨	
K_{SH}	▨	▨	▨				▨	▨	▨	▨						
K_{DATA}	▨	▨	▨				▨	▨	▨	▨						
K_{ALT_OP}	▨	▨	▨				▨	▨	▨	▨						
K_R	▨							▨		▨						
K_{ALT_I}	▨						▨	▨	▨	▨						
K_{INJ}	▨		▨				▨	▨	▨	▨						
K_{SELF_UNP}						▨			▨		▨	▨		▨	▨	
K_{SELF_CIPH}						▨			▨		▨	▨		▨	▨	
K_{RET}	▨						▨	▨	▨							
K_{RET_+}		▨					▨	▨	▨							
K_{MET}							▨		▨			▨		▨	▨	
K_{NSC}													▨		▨	

Table I: Methods coverage evaluation

Table II shows the comparison results of false positive and false negative rates for the above methods. The rate was calculated for those classes of malicious objects, which were covered by the appropriate method. It is important to note the following fact. As table II shows, rates of false positive are low enough. Nevertheless, the number of false positives on the real channels reach very high values, because of the large volume of transmitted data.

Table III shows computational complexity of the methods.

We consider the methods in terms of their applicability to the analysis of traffic on high-speed channels, as well as provide deeper understanding the space of the algorithms, comparison and tradeoffs between them.

For example, it is known that the method **ButterCup** could detect the exploits with many kinds of obfuscation (see table I). But the method usage on real channels is problematic. This is due to the fact that the method uses signatures of the return address, but a static return address in the modern exploits isn't used. In addition, the ButterCup usage as the only one detection method implies a large number of false positives. Nevertheless, the method can be used as an additional check with other tools, as it doesn't require much time and computing costs. The method can be applied to channels with any traffic profile (with any probability of executable code will appear in the channel), as well as permits analysis of high-speed data in real time. Average throughtput of the method, calculated analytically, is $4,34Mb/s$.

Both **Polygraph** and **Hamsa** have similar pre-processing requirements. Both of these methods are based on the automatic generation of context-dependent signatures and provide similar shellcode classes coverage. Nevertheless, the method Hamsa isn't suited for polymorphic versions of the virus detection because of specifics of generated signatures. Different kinds of Polygraph's signatures provide a more flexible method. Although, the polymorphic version of the virus isn't detected by Polygraph in general case. All three Polygraph's signature classes have advantages and disadvantages. The token-subsequence signatures are more specific than the equivalent conjunction signatures. However, some exploits may contain invariants that can appear in any order. In that case, the token-subsequence signatures are more preferable. The Bayes signatures are generated more quickly than the others and are more useful when the invariants arise in exploits some of the time. The authors recommend to use all three types of signatures at the same time, but it implies a large overhead. For example, Polygraph in the best case 64 times slower than the Hamsa algorithm, in the worst case this value reaches 361 times. Average throughput of the Hamsa, estimated analytically, is $7,35Mb/s$ which makes the method applicable in real-time analysis of high-speed traffic. Average throughput of Polygraph without clustering reaches the value of $10Mb/s$, but the accuracy of the method decreases in the same time. Average throughput of the method with clustering reaches the value of $0.04Mb/s$ only. In that case method can be used only as off-line analyzier.

In contrast to the Hamsa and Polygraph, the **Structural analysis** method cam detect some types of obfuscated shellcode. Moreover, in some cases it is able to detect

Method	FP, %	FN, %	Testing sets
Buttercup	0.01	0	TCPdump files of network traffic from the MIT Lincoln Labratory IDS evaluation Data Set
Hamsa	0.7	0	Suspicious pool: Polygraphs pseudo polymorphic worms; polymorphic version of Code-Red II; polimorphic worms, created with CLET and TAPiON; Normal traffic: HTTP URI
Polygraph	0.2	0	Malicious pool: the Apache-Knacker exploit, the ATPhttpd exploit, BIND-TSIG exploit; Network traces: 10-day HTTP trace (125,301 flows); 24-hour DNS trace
Stride	0.0027*	0	Malicious pool: sleds, generated by the Metasploit Framework v2.2; Network traffic: HTTP URI;
Racewalk	0.0058	0	Malicious pool: sleds, generated by the Metasploit Framework v2.2; Normal traffic: HTTP URI, ELF executables, ASCII text, multimedia, pseudo-random encrypted data.
Styx	0	0	Malicious pool: exploits generated using the Metasploit framework; Normal data: network traffic collected at a enterprise network, which is comprised mainly of Windows hosts and a few Linux boxes.
Structural	0.5	0	Malicious pool: malicious code that was disguised by ADMmutate; Normal traffic: data consists to a large extent of HTTP (about 45%) and SMTP (about 35%) traffic The rest is made up of a wide variety of application traffic: SSH, IMAP, DNS, NTP, FTP, and SMB traffic.
SigFree	0**	0	Malicious pool: unencrypted attack requests generated by Metasploit framework, worm Slammer, CodeRed Normal data: HTTP replies (encrypted data, audio, jpeg, png, gif and flash).
STILL	0**	0	Malicious pool: code that was generated using Metasploit framework, CLET, ADMmutate
Semantic aware	0	0	Malicious pool: set of obfuscated variants of B[e]agle; Normal data: set of 2,000 benign Windows programs
Emulation	0.004	0	Malicious pool: code generated by Clet, ADMmutate, TAPiON and Metasploit framework; Normal data: random binary content
HDD	0.0126	0	Malicious pool: code generated by Metasploit Framework, ADMmutate and Clet; Normal data: UDP, FTP, HTTP, SSL, and other TCP data packets; Windows binary executables
NSC	0	0	Malicious pool: code generated by Avoid UTF8/tolower, Encoder and Alpha2 Normal data: three different kinds of random content such as binary data, ASCII-only data, and printable-only characters
IGPSA	0	0	Malicious pool: code generated by Clet, ADMmuate, Jempiscodes, TAPioN, Metasploit Framework Normal data: two types of traffic traces: one contains common network applications HTTP and HTTPs, of port 80 and 443; the other contains traces of port 135, 139 and 445
PolyUnpack	0	0	Malicious pool: 3,467 samples from the OARC malware suspect repository.
APE	0	0	Malicious pool: IIS 4 hack 307, JIM IIS Server Side Include overflow, wu-ftpd/2.6-id1387, ISC BIND 8.1, BID 1887 exploits; Normal data: HTTP and DNS requests.

Table II: Accuracy of the methods. FP stands for “False Positives” and FN stands for “False Negatives”

Method	Complexity	Remarks
Buttercup	$O(N)$	N is the lenght of S
Hamsa	$O(T \times N)$	N is the lenght of S, T is the number of tokens in signature
Polygraph	$O(N)$ $O(N + S^2)$ $O(M^2 \times L)$	without clusters N is the lenght S with clusters S is the number of clusters method's training M the lenght of malware training information L - the lenght of legitimate training information
Stride	$O(N \times l^2)$	N is the lenght of S, l is the lenght of NOP-sled
Racewalk	$O(N \times l)$	N is the lenght of S, l is the lenght of NOP-sled
Styx	$O(N)$	N is the lenght of S
Structural	$O(N)$	N is the lenght of S
SigFree	$O(N)$	N is the lenght of S
STILL	$O(N)$	N is the lenght of S
Semantic	$O(N)$	N is the lenght of S
Emulation	$O(N^2)$	N is the lenght of S
HDD	$O(N + K^2 \times T^2)$	N is the lenght of S K is the number of suspicious chains T is maximum lenght of suspicious chains
NSC	$O(N^2)$	N is the lenght of S
IGPSA	$O(N^2)$ $O(CN)$	non-optimized optimized
PolyUnpack	$O(N)$	N is the lenght of S
APE	$O(N \times 2^l)$	N is the lenght of S l is the lenght of NOP-sled

Table III: Methods complexity

metamorphic shellcode as it generates program's structure dependent signatures. In spite of the fact the algorithmic complexity of all three algorithms is comparable (see table III), Structural analysis slower than the others. Because of the time complexity of algorithm, traffic analysis is possible in off-line mode only. Average throughtput of the method reaches the value of $1Mb/s$. In addition,technique cannot detect malicious code that consists of less than k blocks. That is, if the executable has a very small footprint method cannot extract sufficient structural information to generate a fingerprint. The authors chose 10 for k in their experiments.

The **Racewalk** method improves the **Stride** algorithm by significally reducing of computational complexity. Both Racewalk and Stride can be used in real-time analysis of high-speed channels. When comparing the methods of false positives rate it is necessary to consider the following observation fo the Stride algorithm. There is a possibility that NOP-equivalent byte sequence can occur in legitimate traffic. For example, a sequence of bytes may appear as part of ELF executable, ASCII text, multimedia or pseudo-random encrypted data. Thus, the value presented in Table II for this type of legitimate traffic may vary from what is represented. Both of these methods significantly exceed the speed of the **APE** method of abstract interpretation which also detects NOP-sled. In that case it is diffucult to use APE on real channels.

The **Styx** method is able to detect self-unpacked and self-ciphered shellcode. Nevertheless, in the average case Styx is slower than similar methods of dynamic analysis. Particularly, the average throughput of the method is $0.002Mb/s$. That significantly decreases the method's applicability. Nevertheless, it can be used as a supplement to other shellcode detection algorithms. The method as an additional tool to others can increase the shellcode space coverage. Another considered method which is based on CFG construction is **Semantic aware algorithm**. It is also characterized by low-speed analysis. In that case the method cannot be used in real-time mode even on channels with low bandwidth. The second limitation of method comes from the use of def-use chains.The def-ude relations in the malicious template effectively encode a specific ordering of memory updates. Thus, the algorithm can detect only those program that exhibit the same ordering of memory updates. Nevertheless, the method can be used as additional checking tool to others shellcode detection algorithms.

Methods **SigFree** and **STILL** together providing particularly complete coverage of all shellcode classes. In addition, methods are able to work in real-time mode on high-speed channels. However, the value of false positives rates of SigFree and STILL methods represent only the traffic profile, which does not allow any kind of executables. For the other traffic profile false positive rates of these methods are extemely high. That fact decreases the aplicability of SigFree and STILL.

Significant advantage of methods **Emulation, NSC Emulation, IGPSA** is their resistance to anti-static evasion techniques. At the same time, all these methods have a limitated applicability since they can detect only shellcode classes that contain anti-static obfuscation. As example, the Emulation method 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 algorithm. However, polymorphic engines are becoming more prevalent and complex. The method's throughput is analytically evaluated as $1Mb/s$. Method NSC Emulation, running at average throughput $1.25-1.5Mb/s$ is focused on finding non-self-contained shellcode which practically doesn't occur in real traffic. Thus, the applicability of the method isn't clear. The average throughput of IGPSA algorithm is $1.5Mb/s$/ Algorithms IGPSA and Emulation can interchanged with each other.

Average estimated throughput of the hybrid method **HDD** is $1.5Mb/s$. That allows to use the method on the channels characterized by a relatively low bandwidth in real-time mode. An important advantage of the method is its ability to detect metamorphic shellcode, along with other classes that use anti-static obfuscation techniques. However, the authors didn't test the method on non-exploit code that uses code obfuscation. code encryption, and self-modification. That fact can potentially change the false positives rate proposed by the authors. Thus, this is true for the other methods which detects polymorphic and metamophic shellcodes.

The throughput of **PolyUnpack** hybrid method is significantly lower than HDD and estimated as $0.05Mb/s$. This is due to time requirement to model generation and long delays between running program request and model responce. In addition, with decreasing of the program size, the throughtput of method desreases respectively. Nevertheless, the method characterized by 100% detection accuracy and zero false positives rate. That makes possible to use method as an additional analyzer to other shellcode detection algorithms.

IV. Proposed approach and conclusion

This paper discusses techniques to detect malicious executable code in high-speed data transmission channels. Malicious executable code is characterized by a certain set of features by which the entire set of malware can be divided into the classes. Thus, the problem of shellcode detection can be formulated in terms of recognition theory. Each shellcode detection method can be considered as a classifier which assigns the executable malicious code to one of the classes K_i of shellcode space. Each classifier has its own characteristics of shellcode space coverage, false negative and false positive rates, computational complexity.

Using the set of classifiers we can formulate the problem of automatic synthesis of such hybrid shellcode detector,

PDF-XChange Viewer
Click to buy NOW!
www.docu-track.com

which will cover all shellcode feature classes and reduce the false positive rates while reducing the computational complexity of the method compared with the simple linear combination of algorithms. The method should be synthesized in conformance with the profile of traffic channel data. In other words, the method should consider the probability of executable code transmission through the channel, etc. Let us consider the problem of algorithm synthesis as construction of a directed graph $G = (V, E)$ (see Fig. 3) with a specific topology, where $\{V\}$ is the set of nodes which are classifiers themselves, $\{E\}$ is the set of arcs. Each arc represents the route of flow data. We decided to include in the graph such classifiers (methods) that provide the most complete coverage of the shellcode classes $K_1, \dots, K_l$. Each of the selected classifiers is assigned with two attributes: false positive rates and complexity. The attributes' values can be calculated by profiling, for example.

This qualifier must change the corresponding bit in the information vector from the delta to 0 or 1. If the corresponding bit different from the delta, the classifier produces for him a logical or operation.

Each arc (v_i, v_j) is marked with one of the classes K_r if v_i classifier checks whether the object (flow data) belongs to class K_r. The v_i classifier changes the corresponding bit $\alpha_r(S)$ in the information vector $\tilde{\alpha}(S) = (\alpha_1(S), \alpha_2(S), \dots, \alpha_l(S))$ from Δ to value from $\{0, 1\}$. If $\alpha_r(S) \neq \Delta$ then the classifier produces for it a logical `or` operation: $\alpha_r(S) = \alpha_{r_{CURRENT}}(S) \mid\mid \alpha_{r_{PREVIOUS}}(S)$. If the classifier v_i checks whether the object S belongs to several classes of shellcode space, then the vertex v_i has several outgoing arcs with the corresponding notes. Similarly, the classifier changes the values of corresponding bits in information vector $\tilde{\alpha}(S)$. In addition, if vertex v_i has several incoming arcs, then the results of classifiers, from which the arcs are outgoing, merge with each other.

We assume that each node is associated with the type of the set $\{REDUCING, NON\_REDUCING\}$. If a node v_i has type $REDUCING$, then if the classifier v_i concludes object S to be legitimate, the flow is not passed on. That implies the computational cost decreases and input flow is reduced. The reduced flow example is shown in Fig. 4

We associate each path in the graph G with its weight. The weight consists of a combination of two parameters: i) the total processing time, and ii) the false positive rates. it is necessary to include a classifier with lowest false positive rates to each path in G.

As part of the problem being solved it is necessary to propose a topology of graph G such that: i) the traffic profile will be taken into account; ii) all pathes will be completed in the shortest time, and iii) all pathes will be completed with the lowest false positive rates. We will consider that problem in terms of multicriteria optimization theory.

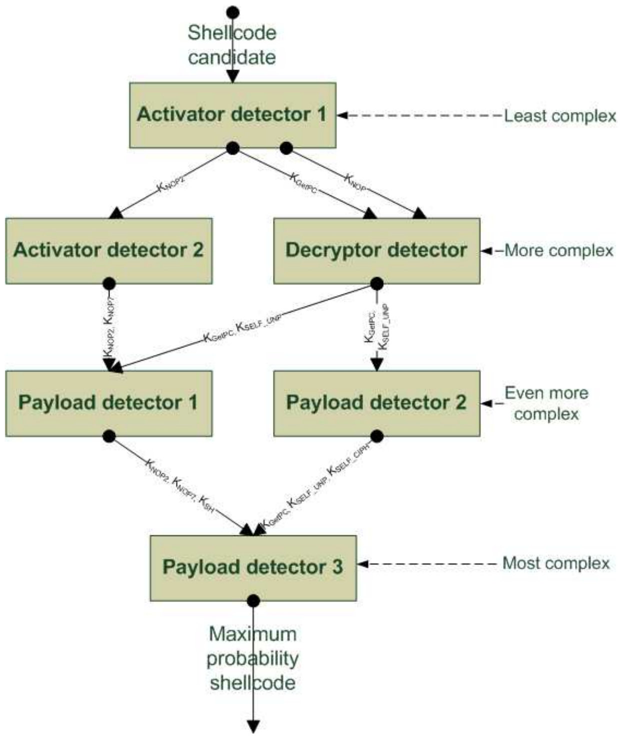

Figure 3: Graph example. Solid arrow represents the route of shellcode candidates. The arc (v_i, v_j) is marked with one of the classes K_x if v_i classifier checks whether shellcode candidate belongs to class K_x.

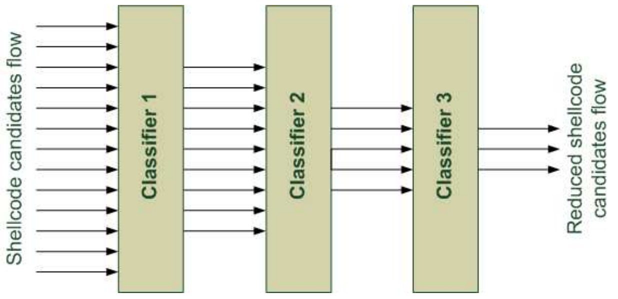

Figure 4: Example of flow reducing. Arrows represent the flow of shellcode candidates. The Classifiers 1, 2 and 3 consider part of the objects as legitimate, so they are not passed on.

REFERENCES

[1] Y. I. Zhuravlev, *Algebraic approach to the solution of recognition or classification problems*. Pattern recognition and image analysis, 1998, vol. 8; no.1, 59-100.

[2] Team Cymru *Malware Infections Market*. [PDF] http://www.team-cymru.com/ReadingRoom/Whitepapers/2010/Malware-Infections-Market.pdf

[3] B. Stone-Gross et al., *Your Botnet is My Botnet: Analysis of a Botnet Takeover*. Technical report, University of California, May 2009.

[4] K. Kruglov, *Monthly Malware Statistics: June 2010. Kaspersky Lab Report, June 2010.* [HTML] http://www.securelist.com/en/analysis/204792125/Monthly_Malware_Statistics_June_2010

[5] P. Porras, H. Saidi, V. Yegneswaran, *An Analysis of Conficker's Logic and Rendezvous Points*. Technical Report, SRI International, Feb 2009.

[6] FBI, *International Cooperation Disrupts Multi-Country Cyber Theft Ring*. Press Release, FBI National Press Office, Oct 2010.

[7] U. Payer, M. Lamberger, P. Teufl, *Hybrid engine for polymorphic shellcode detection*. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA05). Berlin: Springer-Verlag, 2005. 19-31

[8] R. Chinchani, E. Berg, *A fast static analysis approach to detect exploit code inside network flows*. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05). Berlin: Springer-Verlag, 2005. 284-308

[9] C. Kruegel, E. Kirda, D. Mutz, et al., *Polymorphic worm detection using structural information of executables*. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05). Berlin: Springer-Verlag, 2005

[10] P. Akritidis, E. Markatos, M. Polychronakis, and K Anagnostakis, *Stride: Polymorphic sled detection through instruction sequence analysis*. In Proc. of the 20th IFIP International Information Security Conference (SEC'05), 2005.

[11] D. Gamayunov, N. T. Minh Quan, F. Sakharov, E. Toroshchin *Racewalk: fast instruction frequency analysis and classification for shellcode detection in network flow* In: 2009 European Conference on Computer Network Defense. Milano, Italy, 2009

[12] A. Pasupulati, J. Coit, K. Levitt, et al., *Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities*. In: Proceedings of Network Operations and Management Symposium 2004. Washington: IEEE Computer Society, 2004

[13] E. Filiol, *Metamorphism, formal grammars and undecidable code mutation*. International Journal of Computer Science,2, 2007

[14] Z. Li, M. Sanghi, Y. Chen, et al., *Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience*. In: Proceedings of 2006 IEEE Symposium on Security and Privacy (S&P'06). Washington: IEEE Computer Society, 2006. 32-47

[15] J. Newsome, B. Karp, D. Song, *Polygraph: automatically generating signatures for polymorphic worms*. In: Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05). Washington: IEEE Computer Society, 2005. 226-241

[16] M. Weiser, *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979

[17] X. Wang, C. C. Pan, P. Liu, S. Zhu, *Sigfree: A signature-free buffer overflow attack blocker*. In 15th Usenix Security Symposium, July 2006

[18] X. Wang, Y. Jhi, S. Zhu, *Protecting Web Services from Remote Exploit Code: A Static Analysis Approach* In Proc. of the 17th international conference on World Wide Web (WWW'08), 2008.

[19] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, *Semantics-aware malware detection*. In Proc. of 2005 IEEE Symposium on Security and Privacy (S&P'05), 2005.

[20] D. Detlefs, G. Nelson, J. B. Saxe *Simplify: A Theorem Prover for Program Checking*

[21] R. E. Bryant, S. k. Lahiri, S. A. Seshia, *Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions*. In: CAV 02: International Conference on Computer-Aided Verification

[22] A. Stavrou, M. E. Locasto, Y. Song, *On the Infeasibility of Modeling Polymorphic Shellcode* In Proc. of the 14th ACM conference on Computer and communications security (CCS'07), 2007.

[23] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, *Network-level polymorphic shellcode detection using emulation*. In:Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Berlin: Springer-Verlag, 2006

[24] Q. Zhang, D. S. Reeves, P. Ning, et al., *Analyzing network traffic to detect self-decrypting exploit code*. In: Proceedings of the 2nd ACM Symposium on InformAtion, Computer and Communications Security, New York: ACM, 2007. 4-12

[25] L. Wang, H. Duan, X. Li, *Dynamic emulation based modeling and detection of polymorphic shellcode at the network level* Science in China Series F: Information Sciences Volume 51, Number 11, 1883-1897.

[26] M. Polychronakis, K. G. Anagnostakis, E. P. Markatos *Emulation-based Detection of Non-self-contained Polymorphic Shellcode* In Proc. of the 10th international conference on Recent advances in intrusion detection (RAID'07), 2007.

[27] P. Royal, M. Halpin, D. Dagon, R. Edmonds, W. Lee, *PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware* In: Computer Security Applications Conference (ACSAC'06), 2006.

[28] T. Toth, C. Kruegel, *Accurate Buffer Overflow Detection via Abstract Payload Execution* In Proc. of the 5th international conference on Recent advances in intrusion detection (RAID'02), 2002.