Specification and Evaluation of Polymorphic Shellcode Properties Using a New Temporal Logic

Mehdi Talbi, Mohamed Mejri, and Adel Bouhoula

No Institute Given

Abstract. It is a well known fact that polymorphism is one of the greatest find of malicious code authors. Applied in the context of Buffer Overflow attacks, the detection of such codes becomes very difficult. In view of this problematic, which constitutes a real challenge for all the international community, we propose in this paper a new formal language (based on temporal logics such as CTL) allowing to specify polymorphic codes, to detect them and to better understand their nature. The efficiency and the expressiveness of this language are shown via the specification of a variety of properties characterizing polymorphic shellcodes. Finally, to make the verification process automatic, this language is supported by a new IDS (Intrusion Detection System) that will also be presented in this paper.

Key words: Polymorphic Shellcodes, Formal Methods, Temporal Logics, Intrusion Detection.

1 Introduction

Statistics provided by the NVD (National Vulnerability Database) [7] and based on US-CERT alerts [14] reveal that the most common vulnerabilities are Buffer Overflow. Intuitively, a Buffer overflow is a kind of programming error that occurs when a physical input (memory space) receives a large and unexpected value. Attacks that exploit such vulnerabilities, which are mainly due to programmer's carelessness, can cause serious problems. For example, they can cause the crash of a machine (Denial of Service) and, even worse, they can lead the attacker to get the complete control of the targeted machine. The most famous example of the Buffer Overflow problem is related to the Ariane 5 [33] space launcher that crashed shortly after the takeoff in 1996. It was discovered later that the origin of the problem was due to a 16-bit program that could not recognize and use a 64-bit input (Integer Overflow).

Several techniques and tools [10, 19, 22] have been proposed during last years to detect Buffer Overflow attacks. In spite of their great contribution in this field, they are still far from solving the problem. This is due to the subtlety and the complexity of the problem on the one hand, and the remarkable evolution of tools and techniques allowing to discover and exploit these kind of flaws on the other hand. Flawfinder [2], Rats [11], and Retina [12] are examples of source code scanner allowing to detect security flaws. Available platforms such as Nessus [8]

and Metasploit [5] allow to perform various intrusion tests. However, the real revolution in the world of Buffer Overflow attacks comes undoubtedly from the techniques of shellcode obfuscation, which are more and more sophisticated. Issued from the viral scene, polymorphism (payload encoding) is the last find of malicious code authors and can be applied to shellcodes as well. As a result, the malicious code is different at each time making obsolete pattern matching based techniques. Clet team claim that their polymorphic shellcode engine [24] can even defeat Data Mining based IDSs. The reading of [29], which deals with new categories of worms, that mutate and take form according to statistics elaborated on the flow analysis, gives some shivers.

Most of recent systems [16, 40, 44] proposed to detect polymorphic shellcodes are based on some observations (particular patterns and/or behaviors), and the whole detection mechanism is based on these observations. Their interaction with end-users are reduced to the definition of a few parameters limiting their detection capabilities. Formal language-based detection approaches allow to overcome these limitations. Indeed, in such approach, the end-user can express his own observations. In this paper, we propose a new formal language (based on temporal logics) for the specification of a large variety of properties characterizing polymorphic shellcodes. Our intention is to verify these properties against a model (traffic abstraction). More precisely, we want to check the satisfaction relation $\mathcal{M} \models \phi$, where $\mathcal{M}$ is a model representing an abstraction of the audit trail, and ϕ a formula characterizing a polymorphic shellcode property. The simplicity, the expressiveness, and the efficiency of the proposed language are shown via different examples given in this paper. This language is also supported by an IDS prototype making the detection step automatic as it will be shown in this paper.

The remainder of this paper is organized as follows: Section 2 describes Buffer Overflow exploits and presents obfuscation techniques allowing to hide malicious payload in Buffer Overflow attacks. Section 3 presents the model against which the properties will be checked. Section 4 introduces LTL and CTL temporal logics and presents the syntax and semantics of the proposed logic. Section 5 presents properties characterizing polymorphic shellcodes. Section 6 describes the IDS-Logic prototype. Section 7 evaluates the efficiency of the proposed properties to detect polymorphic shellcodes. Section 8 discusses related work, and finally, some concluding remarks on this work are ultimately sketched as a conclusion in Section 9.

2 Polymorphic Shellcodes

In this section, we briefly present Buffer Overflow attacks together with common obfuscation techniques used to hide the malicious payload and specially polymorphic techniques (payload encryption).

2.1 Exploiting Buffer Overflow Vulnerabilities

Buffer Overflow attacks were popularized by Aleph1 in [17]. In this section, we explain how to exploit Buffer Overflow vulnerabilities through the vulnerable code example (*vuln.c*) given hereafter:

```
...
int func(char *srcbuff)
{
char destbuff[100];
strcpy(destbuff, srcbuff);
return 0;
}
...
```

In *vuln.c*, the *strcpy()* function does not check if the destination buffer (*destbuff*) is big enough to contain the data of the source buffer (*srcbuff*). Therefore, *destbuff* can be overflowed. To avoid this, it is sufficient to replace *strcpy()* function by its alternative form *strncpy()*.

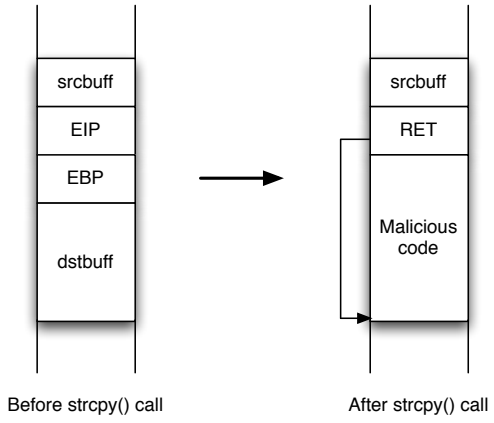


Fig. 1. Memory Organization

Fig. 1 represents the program *vuln* in memory before and after the call of the *strcpy()* function. In order to exploit *vuln* program, one can inject in memory a malicious code (e.g. shellcode /bin/sh) and overwrite EIP pointer with a return address (RET) that points to the beginning of the malicious code. Thus, when the function *func(char *)* returns, the execution flow is redirected to the malicious code: the shellcode is executed. Note that in a remote Buffer Overflow attacks, we don't know exactly the memory address where the malicious code is stored. However, this address can be estimated by reproducing the same environment

according to the target machine (OS/Architecture/Verion of the vulnerable application). In order to compensate the return address estimation error, a sequence of **nop** instructions (0x90) is added to the beginning of the payload. A **nop** instruction performs a null operation. Thus, if the execution flow is redirected to the NOP section, these instructions will be executed until they reach the malicious code. Also, in order to ensure that the EIP pointer is overwritten with the estimated value, the RET address is repeated several times after the malicious code. The malicious payload has therefore the structure given in Fig. 2.

Fig. 2. Classical Payload

Note that a long sequence of **nop** instructions, a repeated return address, and some byte-sequences in shellcodes (e.g. string "/bin/sh") are characteristics of Buffer Overflow attacks, and therefore it is trivial for an IDS to detect the malicious payload given in Fig. 2. However, malicious code authors have developed several techniques allowing to obfuscate the malicious payload and in particular to hide the shellcode.

2.2 Shellcode Obfuscation Techniques

In this section, we present common obfuscation techniques used by hackers, and explain how these techniques can be applied to build up polymorphic shellcodes.

Obfuscation Techniques.

- Instruction Insertion: consists in inserting junk instructions/bytes between the relevant part of the shellcode.
- Code Transposition: consists in changing the order of instructions.
- Register Renaming: consists in changing the name of the used registers. For instance, **inc %eax** and **inc %ebx** have different opcodes, and therefore replacing **eax** by **ebx** will produce different byte-sequences in shellcodes.
- Instruction Substitution: consists in replacing some instructions by their semantically equivalent ones. For instance, the instruction **add %eax, 2** can be substituted with two instructions **inc %eax**. Note that for some particular instructions these substitutions are not possible. For instance, the instruction **int** has no equivalent.
- Alphanumeric Instructions: consists in using ASCII opcodes. Non-alphanumeric instructions in an alphanumeric communication is suspicious. Alphanumeric shellcodes [39] are useful for hackers when the target service accepts only alphanumeric inputs.

- Polymorphism: consists in ciphering the shellcode and attached to it a decipher routine which is different from one attack to another (e.g. by applying metamorphism techniques such as instruction substitution and register renaming). Thus, when the malicious code is executed, the decipher routine is launched first to recover the original form of the shellcode and then the control is given to this code. The literature records many automatic and polymorphic shellcode engines. The most popular are ADMmutate [28], Clet, JempiScode [41], and those proposed by the Metasploit framework.

Polymorphic Shellcode Anatomy. Issued from the viral scene, polymorphism is the last find of malicious code authors and can be applied to shellcodes as well. As a result, the malicious code is different at each time making obsolete pattern matching based techniques. In the case of Buffer Overflow attacks, a polymorphic payload has generally the structure given in Fig. 3.

Fig. 3. Polymorphic Payload

A polymorphic payload is made up of the following parts:

- FAKE_NOP (NOP replacement instructions): instead of **nop** instruction, one can use any other one-byte instruction which has no significant effect (e.g. **inc %eax**, **dec %eax**). The only constraint is to reach the decipher routine without errors. The FAKE_NOP list used by ADMmutate engine contains about fifty instructions. Clet engine uses alphanumeric instruction list to build up the FAKE_NOP section. However, these lists do not present a large range of choices, and FAKE_NOP can be easily detected by any IDS maintaining similar lists. One solution is to extend these lists with several-bytes instructions that satisfy the following condition: suffix and arguments of instructions must be instructions themselves. Thus, it is possible to read valid instructions and reach the decipher routine without errors. The tool ecl-poly [27] allows to generate such FAKE_NOP instructions.
- Decipher Routine: shellcode encryption is based on simple reversible operations (e.g. **add/sub**, **rol/ror**, **xor**). In order to have a completely polymorph code, it is imperative that decipher routine code is different at each time. To achieve this goal, metamorphic obfuscation techniques are used. For example, Clet engine applies instruction substitution and register renaming techniques in order to hide the decipher routine. ADMmutate uses instruction substitution and instruction insertion techniques.
- Padding Zone: this zone is used to pad the empty spaces between the shellcode and the return address. This zone was exploited cleverly by Clet team. It is

about an original idea consisting in filling the padding zone in such a way that the result looks like a normal traffic in terms of probability distribution.

- Return Address: currently, there is only one technique to hide the return address. It consists in varying, from one attack to another, the low-weight bits (1 byte) of the return address.

In [20], we can find several program obfuscation techniques which could be applied to polymorphic shellcodes.

3 Model

One of the basic steps of intrusion detection is the audit trail analysis. It allows to record and analyse some particular actions that have been performed on a system during a period of time. An example of these analyses is to make detection of particular sequence of events characterizing an attack signature.

Almost all related works such as [26, 32] use a trace-based model (i.e. linear model), where a trace is defined as a sequence of events collected from an audit source (e.g. network, host). In the sequel, a trace is denoted by τ, an event is denoted by e and we use the following notations:

If:

$$\tau = e_1, e_2, ..., e_i, e_{i+1}, ..., e_n$$

then :

$$\begin{aligned} \tau[i] &= \text{ event } e_i \\ \tau_i &= \text{ suffix } e_i, e_{i+1}, ...e_n \end{aligned}$$

In case of network source, event e_i represents the i^{th} packet of the trace, where a packet is specified by a set of fields (**protocol**, **dport**, **daddr**, ...) as following:

$$Packet = fd_1, fd_2, \ldots, fd_n$$

with:

$$Packet.fd_i = \text{ content of the field } fd_i$$

Although that the level of traffic abstraction presented above is enough to detect a large variety of attack (e.g. IP Spoofing, Scan, Fragment attacks) using temporal logics (e.g. LTL, ADM), many others attacks require a more detailed model. In particular, to detect some polymorphic shellcodes, a deep analysis of the body field (i.e. *Packet.body*) is necessary. For that reason, we choose to abstract the body field not by a sequence of bits but by a CFG graph (Control Flow Graph) as explained hereafter.

The detection of malicious codes is principally based on the **body** field part. The simple fact of considering the content of this field as a sequence of bits allows us to identify some attacks. For example, we can analyze this field to detect the

return address or some invariant bits sequence between several mutation of a polymorphic shellcode. However, many polymorphic shellcodes can easily escape from being discovered using such analysis. For that reason, several solutions (e.g. APE [44], STRIDE [16], HDE [37]) have recourse to code disassembly to better analyze the nature of malicious codes. As a result, many interesting observations have been made (e.g. invariance of the decipher routine structure). Therefore, in order to take advantage of these observations, we need to refine our trace-based model and specially to refine the representation of the body field content. At first glance, we can be tempted to use a linear model (a sequence of instructions) to represent disassembled code of the body field part. However, this structure has two major drawbacks:

- Malicious code authors can insert junk instructions between the relevant parts of the code and use **jmp** instructions to jump over junk bytes. Junk data allows to hide the shellcode from detection engines which perform linear disassembly.
- Some shellcode obfuscation techniques use **jmp** instructions to jump into the middle of other instructions. This technique well-known as PEX, and issued from the Metasploit framework, is used to hide in particular the decipher routine.

In order to overcome these disadvantages, it is necessary to follow the execution trace. The following of all possible paths (case of conditional jump instructions: **jne**, **jz**, etc.) leads to an arborescent model which corresponds to a CFG graph. Moreover, the statistical study done for Clet, ADMmutate and JempiScode generated code, leads us to the same conclusion as given in [31]. These engines being automatic, it results that some parts of the generated code are presented in the vast majority of the cases according to the same structure (see Fig. 7). So, it is interesting to specify some properties related to such structure. Therefore, we will use an arborescent model rather than a linear one as an auxiliary representation of the body field content. Now, a packet is represented as follows:

$$Packet = fd_1, fd_2, \ldots, fd_n, \mathcal{M}_{\mathcal{CFG}(pos)}$$

where $\mathcal{M}_{\mathcal{CFG}(pos)}$ is an arborescent model associated with the CFG graph. It records events related to packet-data disassembly which is performed from the position *pos*. This choice is motivated by the fact that the content of the body field contains, in the case of malicious code, a data region which doesn't correspond to an executable code (e.g. HTTP header). Disassembly of non-code regions may then have an influence on the CFG graph. The ideal thing would be to perform disassembly from the first instructions of the NOP section. We can consider here, *CFG(pos)* as a function that generates the CFG graph corresponding to the disassembly of the body field content starting from the position *pos*. However, certain properties do not require disassembly process. In order to be in accordance with the given model, a possible solution would be to provide a negative value as an argument to the *CFG()* function (e.g. *CFG(-1)* returns a single empty node). Disassembly operation is potentially costly in intrusion detection. The fact of not having to systemize this process is a considerable advantage.

Formally, a graph can be represented according to several models. Among these are the Labelled Transition System [38] and the Kripke structure [30]. In the first case configuration, emphasis is placed on the actions that the system can do, whereas in Kripke structure, emphasis is placed on the atomic propositions that are true in a state, rather on the actions that allow transitions from one state to another. In our CFG graph, nodes are made up of sequence of instructions. These nodes are interconnected through branch instructions. Therefore, it is more suitable to use the Kripke structure as a model to represent a CFG graph, since the relevant information is associated to nodes rather than to transitions between these nodes. A path π in $\mathcal{M}_{\mathcal{CFG}}$ is a sequence of states:

$$\pi = s_1, s_2, \ldots, s_i, s_{i+1}, \ldots, s_m$$

with $(s_i, s_{i+1}) \in \rightarrow$, for all $i \in \{1, \ldots, m\}$. "$\rightarrow$" is the transition relation of the Kripke structure.

Finally, we will use a linear model to represent the sequence of instructions at node level. The sequence of instructions is defined as following:

$$\sigma = t_1, t_2, \ldots, t_i, \ldots, t_k$$

The event t_i corresponds to the i^{th} instruction. Likewise the case of a packet, an instruction is represented as a set of fields (**inst**, **opcode**, ...).

The proposed model can be summarized as shown by Fig. 4. The $\mathcal{M}_{Packet}$ model represents the sequence of packets. $\mathcal{M}_{\mathcal{CFG}}$ is the model associated with the CFG graph. Finally, $\mathcal{M}_{Inst}$ represents the sequence of instructions at node level.

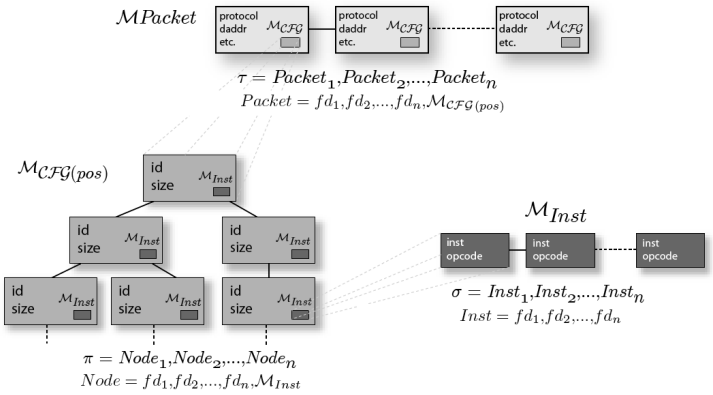


Fig. 4. Model

4 Specification Language

This section introduces a language for the specification and detection of polymorphic shellcodes. Our intention is to propose a formal and appropriate language in

order to specify a large variety of polymorphic shellcode properties, and then to check them against the previously defined model.

Existing formal languages for properties specification [26, 32] are not dedicated to polymorphic shellcodes. In [26], Ben Ghorbel et al. propose to use the ADM logic [15] in order to specify attack signatures. ADM can be viewed as a special variant of μ-calculus modal logic [42]. Initially designed for the specification of electronic commerce properties, it is also appropriate for the intrusion detection issue. In [32], Lesperance et al. use LTL for properties specification. The main shortcoming of these two logics is that their constructs are interpreted over a trace-based model, where a trace is a sequence of packets. As explained in the previous section, this model is not enough complete to describe events related to polymorphic shellcodes. However, we can extend these logics with the ability to specify and check formulae against the proposed model. Compared to LTL, the ADM logic is more complex. Therefore, we will use LTL in order to specify properties and to check them against $\mathcal{M}_{Packet}$ model. Then, we will extend LTL in order to allow the specification of formulae with regard to $\mathcal{M}_{\mathcal{CFG}}$ model. This can be done by the use of an arborescent logic in an embedded manner to LTL. CTL logic is the most appropriate one, as it represents the arborescent version of LTL. This implies a certain homogeneity in the language that we propose. Moreover, Kripke structure, used as model to represent a CFG graph, is often associated with temporal logics such as CTL. Finally, in order to allow the specification of formulae with regard to $\mathcal{M}_{Inst}$ model, we have to extend the CTL logic. This can be done by the use of a linear logic in an embedded manner to CTL. To this end, we will use again the LTL logic. Therefore, the proposed language is based on LTL and CTL temporal logics. Before presenting our logic with its syntax and semantics, let us give a brief introduction to LTL and CTL logics.

4.1 Temporal Logics

Linear Temporal Logic. LTL syntax is defined by the BNF-grammar given by Table 1. The symbols $\neg$ and $\vee$ represent negation and disjunction, respectively. p is an atomic proposition. Finally, X (NeXt) and U (Until) are temporal operators.

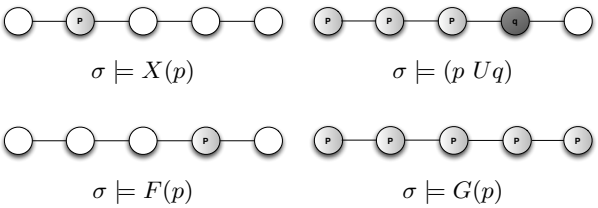
Table 1. LTL Syntax.

$\phi ::= p \mid \neg\phi \mid (\phi_1 \vee \phi_2) \mid X\phi \mid (\phi_1 U \phi_2)$

LTL constructs are interpreted over a sequence of states σ. The formula $X\phi$ is satisfied at a state s_0 in σ if ϕ is satisfied in the next state (s_1). The formula $(\phi_1 U \phi_2)$ is satisfied at a state s_0 in σ if ϕ_2 is satisfied at a state s_k ($k \geqslant 0$), and ϕ_1 is satisfied in all states s_i that precede s_k ($0 \leqslant i \leqslant k$). From temporal operator U, we can derive two useful temporal modalities: $F(\phi)$ (Finally) and $G(\phi)$ (Globally). Their formal definitions are given in Table 6. $F(\phi)$ is satisfied by σ if there exists a state s_k ($k \geqslant 0$) in σ that satisfies ϕ. $G(\phi)$ is satisfied by

σ if ϕ is satisfied at each state in σ. Table 2 gives the intuitive meaning of LTL temporal modalities.

Table 2. LTL Temporal Modalities.



$\sigma \models X(p)$ $\sigma \models (p\ U q)$

$\sigma \models F(p)$ $\sigma \models G(p)$

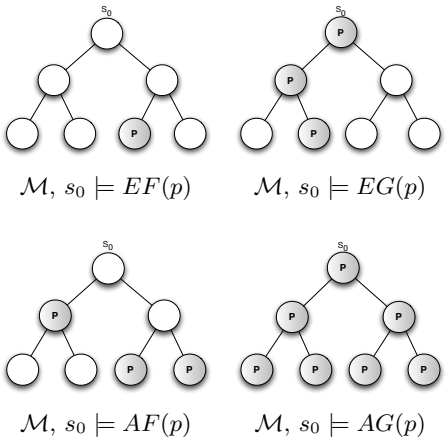
Computational Tree Logic. CTL has the same syntax as LTL with the following additional requirement: temporal operators (X, U, F and G) are prefixed by path quantifiers A (All paths) or E (there Exists a path). CTL syntax is defined by the BNF-grammar given by Table 3. Formulae such as $AF(\phi)$, $AG(\phi)$, etc., can be derived using the abbreviations given in Table 6.

Table 3. CTL Syntax.

$\phi ::= p \mid \neg\phi \mid (\phi_1 \vee \phi_2) \mid EX(\phi) \mid EG(\phi) \mid E(\phi_1 U \phi_2)$

CTL constructs are interpreted over an arborescent model $\mathcal{M}$ (Kripke structure) having as root a state s_0. Table 4 gives the intuitive meaning of CTL temporal modalities.

Table 4. CTL Temporal Modalities.



$\mathcal{M}, s_0 \models EF(p)$ $\mathcal{M}, s_0 \models EG(p)$

$\mathcal{M}, s_0 \models AF(p)$ $\mathcal{M}, s_0 \models AG(p)$

4.2 Proposed Logic

Syntax and Semantics. The syntax of our logic is defined by the BNF-grammar given in Table 5.

Table 5. Syntax.

$\phi ::= [\alpha] \mid [\alpha].\phi \mid (\phi_1 U \phi_2) \mid (\phi_1 \vee \phi_2) \mid \neg\phi$	$\mathcal{M}_{Packet}$
$\alpha ::= P_{packet} \mid \theta \mid (\alpha_1 \vee \alpha_2) \mid \neg\alpha$	$Packet$
$\theta ::= \prec \psi \succ \mid E(\prec \psi \succ .\theta) \mid EG(\theta) \mid E(\theta_1 U \theta_2) \mid (\theta_1 \vee \theta_2) \mid \neg\theta \mid \lfloor\theta\rfloor_w$	$\mathcal{M}_{CFG}$
$\psi ::= P_{node} \mid \beta \mid (\psi_1 \vee \psi_2) \mid \neg\psi$	$Node$
$\beta ::= \{P_{inst}\} \mid \{P_{inst}\}.\beta \mid (\beta_1 U \beta_2) \mid (\beta_1 \vee \beta_2) \mid \neg\beta$	$\mathcal{M}_{Inst}$

We note that:

$\mathcal{L}_\phi = LTL([\alpha])$
$\mathcal{L}_\theta = CTL(\prec \psi \succ)$
$\mathcal{L}_\beta = LTL$

- $LTL[\alpha]$ means that states of the $\mathcal{M}_{Packet}$ model are no longer verified through classical LTL atomic propositions, but through more complex formulae given by α-grammar.
- Similarly for CTL$\prec \psi \succ$, states of the $\mathcal{M}_{\mathcal{CFG}}$ model are verified through formulae specified in ψ-grammar.

We have also introduced some slight modifications to LTL and CTL grammars:

- In the neXt formula, we are not only interested by the successor state, but also by the current state (e.g. formula $[\alpha].\phi$).
- We have introduced a new formula to CTL ($\lfloor\theta\rfloor_w$) that limits CFG graph paths to a depth equal to w. This prevents from a potential infinite loop when checking formulae of type $E(\theta_1 U \theta_2)$.

Note that classical shortcuts such that $\wedge$, $\rightarrow$, $\leftarrow$, $F(\phi)$, $G(\phi)$, etc., can be used with their usual meaning shown in Table 6.

The logic defines 3 types of atomic propositions: P_{packet}, P_{node} and P_{inst} that correspond to **Packet**, **Node** and **Instruction** events (respectively). These propositions have all the same format: **Name Operator Value**. The term **Name** is an element of fields list defined for each event. **Operator** is used for comparison, and **Value** is a simple value or an interval of values.

- **Atomic Proposition** P_{packet}: the set of fields to be checked is defined through the proposition **Name**.

$$Name ::= protocol \mid saddr \mid sport \mid daddr \mid dport \mid flags \mid frags \mid body$$

- **flags** represents the set of TCP flags: **urg**, **ack**, **push**, **rst**, **syn**, and **fin**.
- **frags** represents the set of fragmentation fields: **fo** and **mf**.

Examples:
1. protocol = "tcp"
2. dport = 80, 8080, 8888
3. daddr = 10.10.10.0. . .10.10.10.254
4. flags = 010010 (**syn** and **ack** are set)
5. body = X + "/bin/sh" + Y

In the case of the last proposition, X and Y are variables. The "+" operator enables the concatenation of two strings.

- **Atomic Proposition** P_{node}: we associate to **Node** event two attributes: **id** (node identifier) and **size** (number of instructions in a node).
- **Atomic Proposition** P_{inst}: Intel manual [3] defines the format of instructions. The overall fields which constitute this structure, enables to determinate the nature of the instruction, its parameters and the involved registers. In this study, we define two attributes in order to represent an instruction: **opcode** (instruction code) and **inst** (instruction name).

Examples:
1. inst = "xor", "ror", "rol", "not", "and"
2. opcode = "\x90" (nop instruction)

Table 6. Abbreviation of formulae.

$\bot$	$\equiv \neg\top$
$(\phi_1 \wedge \phi_2)$	$\equiv \neg(\neg\phi_1 \vee \neg\phi_2)$
$(\phi_1 \rightarrow \phi_2)$	$\equiv (\neg\phi_1 \vee \phi_2)$
$(\phi_1 \leftrightarrow \phi_2)$	$\equiv ((\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1))$
$F(\phi)$	$\equiv (\top U \phi)$
$G(\phi)$	$\equiv (\neg F(\neg\phi))$
$AG(\phi)$	$\equiv \neg EF(\neg\phi)$
$AF(\phi)$	$\equiv \neg EG(\neg\phi)$
$EF(\phi)$	$\equiv E(\top U \phi)$
$A(\phi_1 U \phi_2)$	$\equiv (\neg E(\neg\phi_2 U(\neg\phi_1 \wedge \neg\phi_2)) \wedge \neg EG(\neg\phi_2))$
$A(\phi_1.\phi_2)$	$\equiv \neg(E(\neg\phi_1.\phi_2) \vee E(\phi_1.\neg\phi_2))$

The semantics of our logic is given in Table 7. It is derived from LTL/CTL-semantics. Definitions of some notations are given below:

- Satisfaction relations $\models_\phi$, $\models_\alpha$, $\models_\theta$, $\models_\psi$, and $\models_\beta$ are related to formulae ϕ, α, θ, ψ and β (respectively).

- Label functions L_p, L_n and L_i allow to check atomic propositions at packet level, node level and instruction level (respectively).
- The "." operator is used to extract a sub-model from a given event (e.g. $e_0.CFG$ means that $\mathcal{M}_{\mathcal{CFG}}$ model is derived from the event e_0 where e is a **Packet** event).

Table 7. Semantics.

$\mathcal{M}_{Packet}$	
$\tau \models_\phi [\alpha]$	iff $\tau[0] \models_\alpha \alpha$
$\tau \models_\phi [\alpha].\phi$	iff $\tau[0] \models_\phi [\alpha]$ and $\tau^1 \models_\phi \phi$
$\tau \models_\phi (\phi_1 U \phi_2)$	iff $\exists\ j \geqslant 0 \mid \tau^j \models_\phi \phi_2$ and $(\forall\ 0 \leqslant k \leqslant j, \tau^k \models_\phi \phi_1))$
$\tau \models_\phi (\phi_1 \vee \phi_2)$	iff $\tau \models_\phi \phi_1$ or $\tau \models_\phi \phi_2$
$\tau \models_\phi \neg\phi$	iff $\tau \not\models_0 \phi$
Packet	
$e_0 \models_\alpha P_{packet}$	iff $P_{packet} \in L_p(e_0)$
$e_0 \models_\alpha (\alpha_1 \vee \alpha_2)$	iff $e_0 \models_\alpha \alpha_1$ or $e_0 \models_\alpha \alpha_2$
$e_0 \models_\alpha \neg\alpha$	iff $e_0 \not\models_1 \alpha$
$e_0 \models_\alpha \theta$	iff $e_0.CFG \models_\theta \theta$
$\mathcal{M}_{CFG}$	
$\mathcal{M}_{CFG}, s_0 \models_\theta \prec \psi \succ$	iff $s_0 \models_\psi \psi$
$\mathcal{M}_{CFG}, s_0 \models_\theta E(\prec \psi \succ .\theta)$	iff $\mathcal{M}_{CFG}, s_0 \models_\theta \prec \psi \succ$ and exists s_1, successor state of s_0, such that $\mathcal{M}_{CFG}, s_1 \models_\theta \theta$
$\mathcal{M}_{CFG}, s_0 \models_\theta EG(\theta)$	iff exists a path $\pi = s_0 s_1 ...$ and for all $i \in \{0, 1, \ldots\}$ along this path, we have $\mathcal{M}_{CFG}, s_i \models_\theta \theta$
$\mathcal{M}_{CFG}, s_0 \models_\theta E(\theta_1 U \theta_2)$	iff exists a path $\pi = s_0 s_1 ...$ where $\mathcal{M}_{CFG}, s_0 \models_\theta \theta_1 U \theta_2$ along this path (e.g. $\exists\ j \geqslant 0$, such that $\mathcal{M}_{CFG}, s_j \models_\theta \theta_2$ and $(\forall\ 0 \leqslant k \leqslant j$, we have $\mathcal{M}_{CFG}, s_k \models_\theta \theta_1))$
$\mathcal{M}_{CFG}, s_0 \models_\theta (\theta_1 \vee \theta_2)$	iff $\mathcal{M}_{CFG}, s_0 \models_\theta \theta_1$ or $\mathcal{M}_{CFG}, s_0 \models_\theta \theta_2$
$\mathcal{M}_{CFG}, s_0 \models_\theta \neg\theta$	iff $\mathcal{M}_{CFG}, s_0 \not\models_2 \theta$
Node	
$s_0 \models_\psi P_{node}$	iff $P_{node} \in L_n(s_0)$
$s_0 \models_\psi (\psi_1 \vee \psi_2)$	iff $s_0 \models_\psi \psi_1$ or $s_0 \models_\psi \psi_2$
$s_0 \models_\psi \neg\psi$	iff $s_0 \not\models_3 \psi$
$s_0 \models_\psi \beta$	iff $s_0.Inst \models_\beta \beta$
$\mathcal{M}_{Inst}$	
$\sigma \models_\beta \{P_{inst}\}$	iff $P_{inst} \in L_i(\sigma[0])$
$\sigma \models_\beta \{P_{inst}\}.\beta$	iff $\sigma[0] \models_\beta \{P_{inst}\}$ and $\sigma^1 \models_\beta \beta$
$\sigma \models_\beta (\beta_1 U \beta_2)$	iff $\exists\ j \geqslant 0$, $\sigma^j \models_\beta \beta_2$ and $(\forall\ 0 \leqslant k \leqslant j$, we have $\sigma^k \models_\beta \beta_1))$
$\sigma \models_\beta (\beta_1 \vee \beta_2)$	iff $\sigma \models_\beta \beta_1$ or $\sigma \models_\beta \beta_2$
$\sigma \models_\beta \neg\beta$	iff $\sigma \not\models_4 \phi$

Examples. In order to better understand the logic constructs, we give hereafter examples of formulae, and we show how we can verify them against the model

given in Fig. 5. The model, used as example, is built up from a network audit source. It represents sequence of packets and their associated CFG graphs. For the sake of convenience, we have reported in the model only fields of interest (protocol and flags fields).

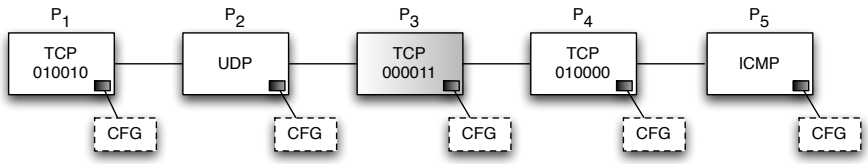


Fig. 5. Model Example

Example 1: Detection of Crafted Packets. Abnormal packets are used by attackers to probe networks or to crash systems. For example, there are several flag combinations that can be classified as abnormal. SYN/FIN is one of these malicious combinations. Indeed, SYN is used to start connection while FIN is used to end an existing one. It is suspicious to have these two flags set in conjunction. To detect such packet, we define the following formula:

$$\phi = F([((protocol = \text{“}tcp\text{”}) \wedge (flags = 000011))])$$

This property allows to verify if there exists a packet ($F([\alpha])$) that satisfies the atomic propositions described by α (TCP packet with the SYN and FIN flags set). It is clear, that the model given in Fig. 5 satisfies the formula ϕ (i.e. $\mathcal{M}_{Packet} \models \phi$), since there exists a packet (P_3) that fulfills the required conditions.

Note that the example given above do not require disassembly, and therefore the CFG graph associated with each packet is made up of a single empty node. In the next example, we show how we can perform a deeper analysis (by disassembling and inspecting CFG graphs).

Example 2: CFG graph inspection. Suppose that disassembly process generates for the packet P_2 the CFG graph given in Fig. 6. Suppose now that we want to verify if the formula ϕ given hereafter is satisfied by our model or not.

$$\begin{array}{l} \phi = F([((protocol = \text{“}udp\text{”}) \wedge \theta)]) \\ \theta = AG(\prec \psi \succ) \\ \psi = F(\{inst = \text{“}int\text{”}\}) \end{array}$$

Formula ϕ allows to verify if there exists an UDP packet ($\in \mathcal{M}_{Packet}$) which associated CFG graph contains at each node an **int** instruction. The first condition (protocol = “udp”) of the formula ϕ is satisfied, since the packet P_2 fulfills this condition. Now, we have to verify if the CFG graph (i.e. $\mathcal{M}_{\mathcal{CFG}}$ model) associated

with packet P_2 satisfies the formula θ. To this end, we need to verify if the formula ψ is "always" satisfied, which is checked thanks to the AG temporal operator. The formula ψ allows to verify if the sequence of instructions (represented by $\mathcal{M}_{Inst}$ model) at node level, contains an **int** instruction. This is verified at each node of the CFG graph associated with packet P_2 (see Fig. 6). Therefore, we conclude that the formula ϕ is satisfied.

Note that, the formula ϕ does not characterize a particular attack. It is defined with the intention to show how we can specify a more elaborated formulae.

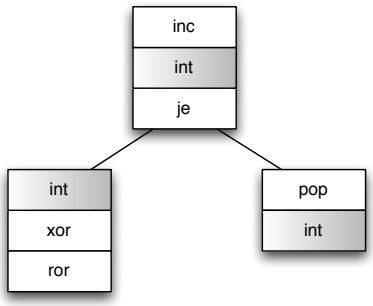


Fig. 6. CFG graph associated with Packet P_2

5 Properties Specification

Now, we are ready to give the specification of properties characterizing polymorphic shellcodes.

5.1 Invariant Byte Sequences Detection

Polymorphic shellcode engines are just at their beginning. Some invariant byte sequences are still observable in the outputs of a given generator. However, these sequences have generally small sizes and they cannot alone characterize a polymorphic code. In the case of Buffer Overflow attacks, the software vulnerability is usually introduced by means of a protocol request. For instance, in order to successfully exploit the "Apache Chunked-Encoding Memory Corruption" vulnerability [1] it is necessary to include in the payload, the header "Transfer-Encoding: chunked". Furthermore, an exploit is generally provided with a return adresses list, which are specific to a given environment (OS/Architecture/Verion of the vulnerable application). The return address can help to define an attack signature. For example, in the case of polymorphic code, the three high-weight bytes of the return address can be used to build attack signature, however the low-weight bits change from one occurrence to another.

Formula given hereafter allows to detect invariant byte sequences in polymorphic Buffer Overflow attacks. As described above, this set comprises invariant exploit framing, invariant overwrite values and invariant substrings in decipher routine. The formula is specific to attacks that exploit the "Apache Chunked-Encoding Memory Corruption" vulnerability in NetBSD systems. It allows to detect malicious payload generated by Clet engine.

$$F([(((protocol = \text{“}tcp\text{”}) \wedge (dport = 80)) \wedge (body = X + \text{“}Transfer-Encoding : chunked\text{”} + Y + \text{“}\backslash x74\backslash x07\backslash xeb\text{”} + Z + \text{“}\backslash xff\backslash xff\backslash xff\text{”} + T + \text{“}\backslash xfa\backslash x0e\backslash x08\text{”} + V))])$$

The flaw "Apache Chunked-Encoding Memory Corruption" is related to HTTP protocol, which explains the first part of the propriety. The second part allows to verify if the body field contains the following patterns:

- "Transfer-Encoding: chunked": invariant exploit framing
- "$\backslash x74\backslash x07\backslash xeb$", "$\backslash xff\backslash xff\backslash xff$" and "$\backslash xfa\backslash x0e\backslash x08$": invariant substrings in Clet's decipher routines.
- "$\backslash xfa\backslash x0e\backslash x08$": three high-weight bytes of the return address 0x080efa00 (overwrite value related to NetBSD systems).

5.2 FAKE_NOP Detection

As described in Section 2, the FAKE_NOP are added at the beginning of the payload to compensate the return address estimation error. The FAKE_NOP section size depends on the vulnerability, but it is generally quite large (more than hundred instructions). A large sequence of consecutive FAKE_NOP instructions is one of the characteristics of polymorphic shellcodes. Several solutions [16][40][44] base their detection on the FAKE_NOP. Thanks to our logic, it is also possible to specify a property allowing to detect FAKE_NOP instructions:

$$F([((protocol = \text{“}tcp\text{”}, \text{“}udp\text{”}) \wedge (\prec G(\{opcode = S\}) \succ \wedge \prec (size \geqslant 100) \succ))])$$

with S = "$\backslash x04$", "$\backslash x05$", "$\backslash x06$", "$\backslash x0c$", "$\backslash x0d$", etc., (set of FAKE_NOP instructions defined in ecl-poly list).

This formula allows to detect if there exists a packet, where the first node of its associated CFG graph contains only instructions belonging to the set S. Note that many tools such as FNORD [40] base their detection on the FAKE_NOP list issued from the source code of ADMmutate engine. This list is not complete and contains only one-byte instructions. In our specification, we use the FAKE_NOP list (set S) defined in ecl-poly tool. This list extends the one used by ADMmutate engine, and includes several-bytes instructions. Thus, the formula given above allows to detect (one/several)-bytes of FAKE_NOP instructions sequence.

Note however, that some Buffer Overflow attacks do not require FAKE_NOP zone. In such attacks, the malicious code is stored in memory above the return address

(above the EIP pointer in Fig. 1). Then, this address is overwritten with a one that points to a **jmp %esp** instruction. Such kind of attacks can be detected by adding in our formula the list of return addresses which point to **jmp %esp** instruction. This list can be extracted, for instance, from the Metasploit database.

5.3 Decipher Routine Detection

CFG graph representation of several instances of polymorphic shellcodes generated by the same engine (e.g. Clet), reveals that the structure of the decipher routine is practically the same for a given input (e.g. /bin/sh shellcode). CFG nodes are interconnected to each other in the same way. Moreover, these nodes often contain the same instruction classes (see Table 9). As we will see in Section 7, 97% of instances generated by Clet engine share the same structure given in Fig. 7. This figure represents the decipher routine structure generated by Clet engine for a given input (/bin/sh shellcode).

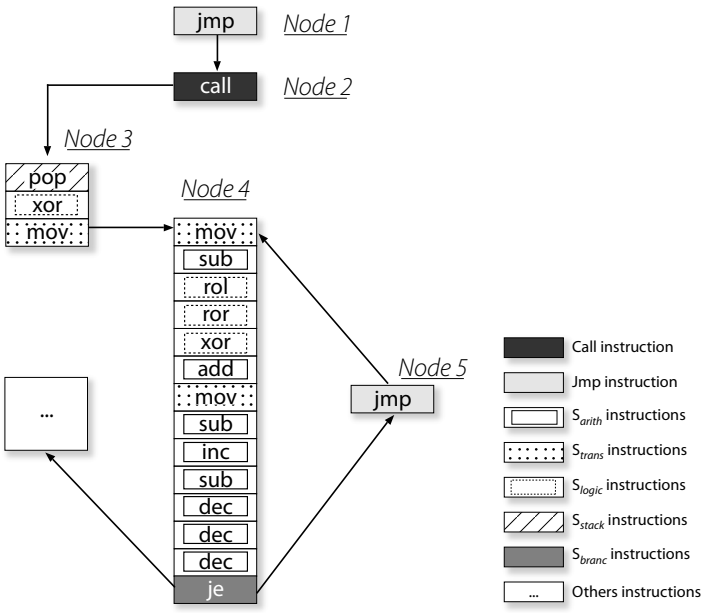


Fig. 7. Example : Decipher routine structure of Clet engine

To detect the decipher routine specific to Clet engine, we define the property given hereafter. Table 8 explains the meaning of each part of the formula.

$F([((protocol = \text{“}tcp\text{”}, \text{“}udp\text{”}) \wedge EF(E(\prec F(\{opcode = \text{“}\backslash xeb\text{”}\}) \succ .$
$E(\prec F(\{opcode = \text{“}\backslash xe8\text{”}\}) \succ .E(\prec ((F(\{inst = S_{arith}, S_{logic}\}) \wedge$
$F(\{inst = S_{trans}\})) \wedge F(\{inst = S_{stack}\})) \succ .E(\prec (((F(\{inst = S_{arith}, S_{logic}\}) \wedge$
$F(\{inst = S_{trans}\})) \wedge F(\{inst = S_{branc}\})) \wedge (id = X)) \succ .$
$E(\prec F(\{opcode = \text{“}\backslash xeb\text{”}\}) \succ . \prec (id = X) \succ)))))))])$

This property allows to verify if **there exists a path** ($\in \mathcal{M}_{\mathcal{CFG}}$) along which we have, **somewhere**, the following sequence of nodes:

1. A node containing the instruction **jmp**,
2. A node containing the instruction **call**,
3. A node containing “stack” instructions (e.g. **pop**), logic/arithmethic instructions (e.g. **xor**) and data transfert instructions (e.g. **mov**),
4. A node containing logic/arithmethic instructions and data transfert instructions. This node ends with conditional branch instruction (e.g. **je**), leading to either node 5, or to node 6 which is the most often invalid (contains invalid instructions), and
5. A node containing the instruction **jmp** leading to the node 4.

More precisely, we are looking for the sequence of nodes given by:

$$\pi_{clet} = Node_1, Node_2, Node_3, Node_4, Node_5, Node_4$$

Table 8. Property characterizing the decipher routine of Clet engine.

Formula	**Comments**
$\phi = F([\alpha])$	Verify if there exists a packet ($\in \mathcal{M}_{Packet}$) that satisfies α
$\alpha = (\alpha_1 \wedge \alpha_2)$	
$\alpha_1 = (protocol = \text{“}tcp\text{”}, \text{“}udp\text{”})$	Only tcp and udp packets are inspected
$\alpha_2 = EF(\theta)$	Verify if there exists a path $\pi \in \mathcal{M}_{\mathcal{CFG}}$ along which θ is satisfied at a given moment in the future
$\theta = E(\prec \psi_1 \succ .E(\prec \psi_2 \succ .E(\prec \psi_3 \succ .$ $E(\prec \psi_4 \succ .E(\prec \psi_5 \succ . \prec \psi_6 \succ)))))$	Verify if the path π contains the sequence of nodes π_{clet}
$\psi_1 = F(p_1)$	Verify if there exists an instruction $\in \mathcal{M}_{Inst}$ (model associated with the 1st node of the sequence π_{clet} (i.e. $\pi_{clet}[1]$)) that satisfies the atomic proposition p_1
$p_1 = \{opcode = \text{“}\backslash xeb\text{”}\}$	jmp instruction (jmp [byte])
$\psi_2 = F(p_2)$	Verify if there exists an instruction $\in \mathcal{M}_{Inst}$ (model associated with $\pi_{clet}[2]$) that satisfies p_2
$p_2 = \{opcode = \text{“}\backslash xeb\text{”}\}$	call instruction (call [dword])
$\psi_3 = ((F(p_3) \wedge F(p_4)) \wedge F(p_5))$	Verify if p_3, p_4 and p_5 are satisfied in $\mathcal{M}_{Inst}$ (model associated with $\pi_{clet}[3]$)
$p_3 = \{inst = S_{arith}, S_{logic}\}$	See table 9
$p_4 = \{inst = S_{trans}\}$	See table 9
$p_5 = \{inst = S_{stack}\}$	See table 9
$\psi_4 = (((F(p_3) \wedge F(p_4)) \wedge F(p_6))$ $\wedge$ $\boldsymbol{(id = X)})$	Verify if p_3, p_4 and p_6 are satisfied in $\mathcal{M}_{Inst}$ (model associated with $\pi_{clet}[4]$)
$p_6 = \{inst = S_{branc}\}$	See table 9
$\psi_5 = F(p_1)$	Verify if p_1 is also satisfied in $\mathcal{M}_{Inst}$ (model associated with $\pi_{clet}[5]$)
$\boldsymbol{\psi_6 = (id = X)}$	Loop Detection. The jmp of node 5 leads to node 4

We define hereafter the properties characterizing the decipher routines of ADMmutate and JempiScode engines, in a similar manner to the Clet engine.

Property characterizing the decipher routine of ADMmutate engine can be specified as following:

$$F([((protocol = \text{“}tcp\text{”}, \text{“}udp\text{”}) \wedge EF(E(\prec F(\{opcode = \text{“}\backslash xeb\text{”}\}) \succ . \\ E(\prec F(\{opcode = \text{“}\backslash xe8\text{”}\}) \succ .E(\prec (F(\{inst = \text{“}xor\text{”}\}) \wedge \\ F(\{inst = S_{trans}, S_{stack}\})) \succ .E(\prec ((F(\{inst = \text{“}xor\text{”}\}) \wedge \\ F(\{inst = S_{arith}\})) \wedge F(\{inst = S_{branc}\})) \succ . \prec F(\{opcode = \text{“}\backslash xeb\text{”}\}) \succ))))))])$$

Property characterizing the decipher routine of JempiScode engine can be specified as following:

$$F([((protocol = \text{“}tcp\text{”}, \text{“}udp\text{”}) \wedge EF(E(\prec F(\{opcode = \text{“}\backslash xeb\text{”}\}) \succ . \\ E(\prec F(\{opcode = \text{“}\backslash xe8\text{”}\}) \succ .E(\prec (F(\{inst = S_{stack}\}) \wedge F(\{inst = S_{arith}\})) \succ . \\ E(\prec F\{inst = S_{arith}\} \wedge F\{inst = S_{branc}\} \succ . \prec F(\{opcode = \text{“}\backslash xeb\text{”}\}) \succ))))))])$$

The previously specified properties are resilient to common obfuscation techniques used by hackers:

- Resilience to instruction substitution (up to a certain limit): the simple fact of replacing, for example, **inc %eax** with **sub $0xffffffff, %eax** will have no impact on formulae, given that **sub**, **inc** $\in S_{arith}$.
- Resilience to instruction insertion: following execution trace during disassembly allows us to exclude junk instructions.
- Resilience to register renaming: the arguments of instructions are not taken into account.
- Resilience to code transposition: the order of instructions at node level is not taken into account.

Table 9. Instruction Classes

Instructions	Class
S_{arith} = inc, sub, add, dec	Arithmetic
S_{logic} = xor, ror, rol, and, not	Logic
S_{trans} = mov, xchg	Data Transfert
S_{stack} = push, pop	Stack
S_{branc} = je, jne, jz, loop, loopne, loopnz	Conditional Branch

Classes of instructions shown in table 9 help to counteract the metamorphism. However, an attacker can use semantically equivalent instructions from different classes. For instance, **mov %eax, %ebx** (class S_{trans}) can be substituted with **push %eax; pop %ebx** (class S_{stack}). For this scenario, the end-user can define

and adapt its own instruction classes according to the different combinations observed in the source code of a given polymorphic engine. Note that there is few substitutions involving instructions belonging to different classes. In certain cases, it is even impossible to replace an instruction with a semantically equivalent one (e.g. **int** instruction).

6 Implementation: The tool IDS-Logic

6.1 Overview

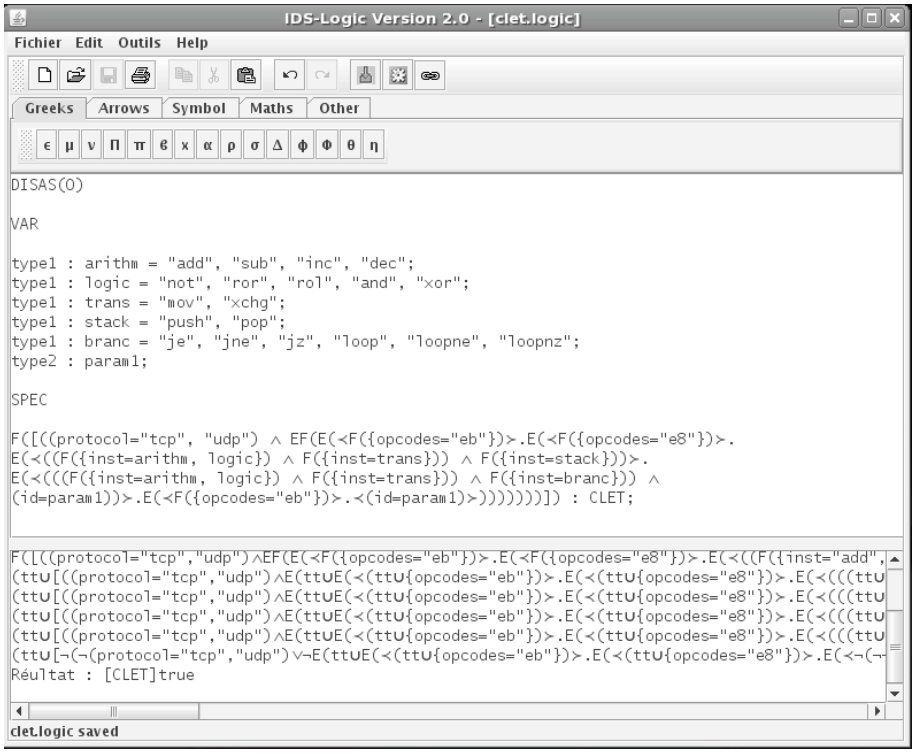


Fig. 8. IDS-Logic Interface.

In this section we give a brief overview of our IDS prototype, written in Java and called IDS-Logic, which is based on the proposed logic. The tool IDS-Logic has a simple and user-friendly interface allowing the end-user to specify his properties and their associated parameters (see Fig. 8). As any language, IDS-Logic provides a compiler that allows mainly to check the syntax of the specified properties. IDS-Logic is also endowed with a model checker allowing to verify the specified properties against a model built up from the audit trail. IDS-Logic is available at [43], from where it is possible to download the source code.

6.2 Architecture

The architecture of our IDS prototype is given in Fig. 9. Hereafter, we give a description of its main components.

Properties Specification. Properties are specified through ".logic" files which have the following structure:

DISAS(pos)
VAR
$type_1\ var_1 = value_1;$
$type_2\ var_2\text{-}var_3\text{-}var_4;$
SPEC
$formula_1 : title_1;$
$formula_2 : title_2;$
...
$formula_n : title_n;$

The proposition **DISAS** is related to disassembly. The clause **VAR** [1] allows the user to define his variables. We define two types of variables: $type_1$ is similar to "#define" in C programming language; variables of $type_2$ are used in atomic propositions (e.g. $protocol = var_1$). Properties are specified through the reserved word **SPEC**. A name can be attributed to a property so that it could be referred later in other formulae or in the output of the analysis.

Compilation: Syntactical Checking. Syntactical checking consists to ensure that the properties are specified while respecting the clauses defined above. It ensures also that the specified formulae are well formed (i.e. properties respect the logic grammar). This procedure is realized by the use of Flex and Bison tools [4].

Properties Transformation. If the specified properties are free of syntax errors, they will be transformed according to the abbreviations given by Table 6. The idea behind these abbreviations is to implement only a few numbers of operators (those of the proposed logic). The rest can be derived from them. For instance, it is not necessary to implement the $\wedge$ operator, since it can be derived from the disjunction ($\vee$) and negation ($\neg$) operators.

Model Construction. This component allows to build up the model defined in Section 3 from an audit file (".net" file). First, the audit file is parsed in order to construct the sequence of packets (i.e. $\mathcal{M}_{Packet}$ model). Then, if disassembly is enabled (i.e. $CFG(pos), pos \geqslant 0$), the body of each packet ($\in \mathcal{M}_{Packet}$) is disassembled in order to construct the CFG graph ($\mathcal{M}_{\mathcal{CFG}}$ model). Note that

[1] Clause VAR is optional

IDSs are prone to real time constraints which result in the number of packets analyzed per second. Disassembly is an expensive operation that can slow down the IDS treatments. For that purpose, it is imperative to find a solution that optimize the disassembly process. The disassembler used in APE allows to solve this problematic. Their disassembler is based on a dictionary of instructions. We follow a similar approach for disassembly.

Execution: Formal Checking. This is the main part of the system. It implements the satisfaction relation "$\models$" of the logic semantics defined in Table 7. The model checker takes a model $\mathcal{M}$ (built up from the audit trail) and a formula ϕ (specified by the end-user), and verify whether $\mathcal{M} \models \phi$ or not. At the end of this stage, the system gives us the detection results: "true" indicates that the analyzed flow contains the attack-evidence described by the specified formula.

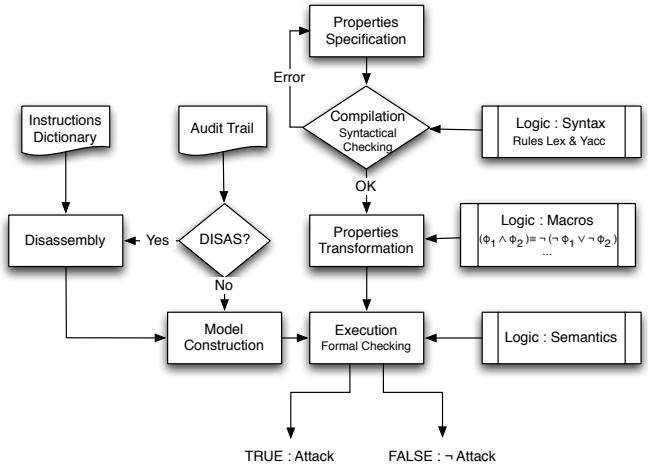


Fig. 9. IDS-Logic Architecture.

7 Properties Evaluation

In this section, we evaluate the efficiency of each property in terms of false positives and false negatives. To this end, we have to define first datasets of network traffic. Publicly available datasets (e.g. MIT Lincoln Lab datasets [6]) are a tremendous asset for the intrusion detection community. However, these datasets have some problems and have been criticized by many researchers. For instance, McHugh pointed out in [34] that the background traffic was generated using too simple models, and if real background traffic was used, it would produce a higher rate of false positives. Moreover, these datasets are not adapted to our study, since we are interested only in polymorphic Buffer Overflow attacks. For all these reasons, we have chosen to define our own datasets[2].

[2] The datasets are available upon request to the authors

7.1 False Positives/True Positives

Datasets. To evaluate our properties in terms of false/true positives, we have defined two datasets.

- HTTP traffic: this first dataset was given to us by the Higher School of Electricity (Supelec Rennes). This evaluation set contains TCPdump data consisting of only HTTP requests (102582 packets). This traffic was collected over two weeks, and was verified to be free of polymorphic attacks. This dataset was used exclusively to evaluate the Invariant Byte Sequences property.
- ELF binaries: we have sent via FTP the content of /bin, /usr/bin, /sbin, and /usr/sbin directories of a Fedora Core 6 Linux distribution (fresh installation). Then, we have captured all the resultant FTP traffic (246157 packets). ELF binaries were sent to a local FTP sevrer (running on the localhost machine), which is connected to any network. Thus, the resultant FTP traffic is free of attacks. This dataset was used to evaluate the properties that require disassembly. The use of such dataset is motivated by the fact that ELF binaries contain executable code, and therefore they are most likely to give false alarms.

Then, we have developed a Java application [3] (using the jpcap package) in order to translate data from TCPdump format to a more simplistic one:

$$protocol \mid saddr \mid sport \mid daddr \mid dport \mid flags \mid frags \mid body$$

Results. Table 10 shows the obtained results. No false positives were reported during the evaluation of the Invariant Byte Sequences Property. The FAKE_NOP property generated many false positives (48 analyzed packets have raised an alarm). This implies that certain solutions which base their detection only on the FAKE_NOP can be inefficient. Finally, no false positives were reported during the evaluation of the Decipher Routine Properties. Several factors can explain this result:

- The defined properties are very specific.
- It is difficult to find ELF binaries that share common structural characteristics (same CFG structure, same instruction classes at node levels) with polymorphic shellcode engines.
- ELF binaries were fragmented into several packets during their transmission via FTP. Fragmentation had led, in certain cases, to the termination of the disassembly process (e.g. address of a control flow instruction is out of bounds).
- Finally, our disassembler does not support the whole instruction set of the Intel manual [3] (e.g. MMX and SIMD instructions are not supported). This perhaps led to invalid instructions during disassembly.

[3] This tool is available at [43]

Table 10. Number of False/True Positives.

Property	False Positive	True Positive
Invariant Byte Sequences	0	102582
FAKE_NOP	48	246109
Clet Decipher Routine	0	246157
ADMmutate Decipher Routine	0	246157
JempiScode Decipher Routine	0	246157

7.2 False Negatives

Datasets. To evaluate our properties in terms of false/true negatives, we have developed some shell scripts allowing to generate sanitized audit files (i.e. ".net" files) in the format of our IDS prototype. The content of these audit files changes depending on the property to be evaluated:

- Invariant Byte Sequences Property: a set of chunked HTTP requests that contain Clet decipher routine.
- Decipher Routine Properties: three sets of packets containing each one 100 outputs generated by Clet, ADMmutate and JempiScode engines (respectively).
- FAKE_NOP Property : a mix of FAKE_NOP generated by ADMmutate and Clet engines.

Results. Table 11 shows the obtained results. No false negatives were reported during the evaluation of the Invariant Byte Sequences Property. This implies that the invariant byte sequences scattered over the malicious code are sufficient to characterize polymorphic codes generated by Clet engine. This illustrates some weaknesses of this engine.

Table 11. Number of False/True Negatives.

Property	False Negative	True Negative
Invariant Byte Sequences	0	100
FAKE_NOP	0	100
Clet Decipher Routine	3	97
ADMmutate Decipher Routine	4	96
JempiScode Decipher Routine	0	100

We obtained a detection rate of 100% for the FAKE_NOP property. This result was expected. Indeed, the property was specified using a FAKE_NOP list that includes all instructions used by ADMmutate and Clet engines. A large number of Clet engine instances (97 out of 100) were detected by the Decipher Routine Property. For the three non-detected instances, the CFG structure was different from the one given by Fig. 7. This was due to the presence of branch instructions

(i.e. S_{branc} instructions) in the node corresponding to the ciphered shellcode (node 6 in Fig. 7). As consequences, some nodes of the CFG graph were splitted into two blocks. In that case, the property is not satisfied any more. Four instances of ADMmutate engine was non-detected for the same reasons. This does not happened during the evaluation of JempiScode Decipher Routine Property, where we have obtained a detection rate of 100%.

8 Related Work

Several preventive techniques have been proposed during the last years to prevent from Buffer Overflow attacks. Some of them tried to act at kernel level (e.g. PAX [10], Openwall [9]) by patching them, others prefer modifying compilers (e.g. StackGuard [22], Stack Shield [13]), while others move toward using dynamic libraries (e.g. Libsafe [19]). Although the significant contributions of these preventive techniques, they can be evaded [18, 21, 23, 45] and the problem is far from being resolved efficiently.

Polymorphic Buffer Overflow attacks are more challenging, and several techniques have been proposed in the literature to detect them. These detective techniques can be classified into two approaches: misuse-based approach and anomaly-based approach. Regarding the former, protection is usually provided by parsing network traffic in order to detect matches with previously defined malicious patterns. Misuse-based IDSs generally produce low false positives rates, but they are not able to detect novel attacks. In contrast, anomaly-based approaches base their detection on a profile of normal network traffic, often built up using Data Mining techniques. Anomaly-based IDSs are able to detect novel attacks, but generally have a high rate of false positives.

Almost all proposed techniques fall into the misuse-detection field. These techniques can be classified into three categories: FAKE_NOP (e.g. FNORD, APE, STRIDE), Invariant Byte Sequences (e.g. Polygraph [35]) and Decipher Routine [31] detection systems.

FNORD, APE and STRIDE base their detection on the FAKE_NOP. FNORD is the most basic one. It maintains a list of one-byte FAKE_NOP instructions (ADMmutate list) and applies pattern matching techniques. It is possible to bypass FNORD by using several-bytes FAKE_NOP instructions. APE and STRIDE are more advanced tools since they perform code disassembly. For instance, APE disassembles the code from randomly chosen positions and reports each time the number of valid instructions: MEL (Maximum Execution Length). If the MEL value is greater than a fixed threshold (MEL > 35), then an alert is raised. APE and STRIDE allows to detect (one/several)-bytes FAKE_NOP instructions. Note however that some Buffer Overflow attacks do not require FAKE_NOP instructions at all, and therefore they cannot be detected by these solutions.

Polygraph uses a set of byte sequences (e.g. exploit framing, overwrite values, invariant substrings in decipher routine), that are common to different instances of a given polymorphic shellcode engine, in order to automatically generate signatures allowing to detect such patterns. Polygraph generates three categories of

signatures: Conjunction signatures, Token-subsequence signatures and Bayes Signatures. Conjunction signatures allow to check if the analyzed flow contains, in any order, the patterns described by these signatures, whereas in Token-subsequence signatures, the order of patterns is taken into account. Finally, Bayes signatures consist in weighting patterns.

Another interesting observation is that there is also structural similarities (e.g. invariance of the decipher routine structure) between mutations of a polymorphic shellcode. Christopher Krügel el al. exploited this fact in [31] in order to automatically generate fingerprints that characterize invariant structures of polymorphic shellcodes. Fingerprints generation involves three processes. First, a polymorphic shellcode is disassembled and abstracted by a CFG graph. Then, all connected k-subgraphs (subgraph with k nodes) are extracted from the CFG graph. Finally, a fingerprint is generated for each subgraph. Fingerprints record how k-subgraph nodes are interconnected to each other, and which classes of instructions (e.g. arithmetic instructions) are contained at each node.

Thanks to the proposed logic, we were able to specify properties which are equivalent to the signatures generated by the detective techniques detailed above. Indeed, properties defined in Section 5 are formal specification of these techniques.

Finally, to conclude with misuse-based approches in the field of polymorphic code detection, we can cite the formal approach proposed in [25]. In this approach, code mutation techniques such as polymorphism and metamorphism are formalized by means of formal grammars. A grammar is defined by an alphabet (e.g. instructions) and a rewriting system. The idea behind this grammar is to define a formal language that represents the different forms that a (polymorhic/metamorphic) code can take with respect to this grammar. The detection technique is based on a language decision problem which consists in determining whether a given mutated code is an instance of the formal language or not.

Regarding anomaly-based approaches, there is only few works [36, 37] that focus mainly on polymorphic shellcodes. In these works, Data Mining methods are used as a learning process which is performed over a set of samples (positives and negatives datasets). For instance, in [37] authors suggest the use of Neural Networks as training process, whereas in [36] authors propose to use the Markov Chains. Note that, Clet engine is endowed with a spectrum analysis mechanism which was designed in order to defeat Data Mining methods. However, evaluation results obtained in [36, 37] show a detection rate of 100% for Clet engine with a low rate of false positives.

9 Conclusions

Buffer Overflow attacks are very powerful. Allied to the polymorphism power, their detection becomes very difficult. In view of this problematic, which constitutes a real challenge, we have proposed in this paper a new formal language allowing the specification of a large variety of properties characterizing polymorphic shellcodes. Simple and expressive, the proposed language is the fruit of temporal

logics (LTL and CTL) combination. This point has been a main goal during the conception phase. Firstly, the simplicity in order to allow the end-user to easily translate his observations into formal specification. Secondly, the expressiveness in order to specify a large variety of polymorphic shellcode properties. In addition, the proposed language is not limited to the characterization of polymorphic shellcodes, but can be used to other ends such as the specification of formulae characterizing a multitude of TCP/IP based attacks (e.g. Crafted Packets, Fragment Attacks). Moreover, the worms spreading over Internet are frequently coded in assembler, and polymorphism/methamorphism techniques are often used by virus authors. The language that we propose can then be used to reflect the behavior of such codes. For example, many worms use the mail as spreading means. These worms contain fragments of code, allowing among others, to get the content of the address book (next targets of the virus). The structure of these fragments are often unchanged for a given virus family. Therefore, the invariant part can be characterized by properties. Finally, in order to validate our work, we have developed an IDS prototype that implements the proposed logic. The results of the evaluation process show a good tradeoff between false positives and false negatives.

References

1. CAN-2002-0392 - apache chunked-encoding memory corruption vulnerability. http://www.securityfocus.com/bid/5033/discuss.
2. Flawfinder. http://www.dwheeler.com/flawfinder.
3. IA-32 intel architecture software developer's manual - instruction set reference. http://www.intel.com/design/pentium4/manuals/index_new.htm.
4. The lex & yacc page. http://dinosaur.compilertools.net/.
5. Metasploit. http://www.metasploit.com/.
6. MIT lincoln laboratory. http://www.ll.mit.edu/.
7. National vulnerability database. http://nvd.nist.gov/statistics.cfm.
8. Nessus. http://www.nessus.org.
9. Opewall. http://www.openwall.com/.
10. PAX. http://pax.grsecurity.net/docs/index.html.
11. Rats. http://www.securesoftware.com.
12. Retina. http://www.eeye.com.
13. Stack Shield. http://www.angelfire.com/sk/stackshield/.
14. US-CERT. http://www.us-cert.gov/.
15. Kamel Adi, Mourad Debbabi, and Mohamed Mejri. A new logic for electronic commerce protocols. *Theoretical Computer Science*, 291(3):223–283, 2003.
16. P. Akritidis, Evangelos P. Markatos, Michalis Polychronakis, and Kostas G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *SEC*, pages 375–392, 2005.
17. Aleph1. Smashing the stack for fun and profit. http://www.phrack.org/issues.html?issue=49&id=14.
18. Christophe Bailleux and Christophe Grenie. Protections contre l'exploitation des débordements de buffer - bibliothèques et compilateurs. http://www.miscmag.com/.
19. Arash Baratloo, Navjot Singh, and Timothy Tsai. Libsafe: Protecting critical elements of stacks.

20. Philippe Beaucamps and Eric Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21, 2007.
21. Bulba and Kil3r. Bypassing Stackguard and Stackshield. http://www.phrack.org/issues.html?issue=56&id=5.
22. Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM*. USENIX Association, 1998.
23. Solar Designer. Getting around non-executable stack (and fix). http://www.securityfocus.com/archive/1/7480.
24. Theo Detristan, Tyll Ulenspiegel, Yann Malcom, and Mynheer Superbus Von Underduk. Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/issues.html?issue=61&id=9.
25. Eric Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):7075, 2007.
26. Meriam Ben Ghorbel, Mehdi Talbi, and Mohamed Mejri. Specification and detection of TCP/IP based attacks using the ADM-logic. In *ARES*, pages 206–212. IEEE Computer Society, 2007.
27. Yuri Gushin. Nids polymorphic evasion - the end? http://www.ecl-labs.org/papers.html.
28. K2. Admmutate. http://www.ktwo.ca/.
29. Oleg Kolesnikov and Wenke Lee. Advanced polymorphic worms: Evading IDS by blending in with normal traffic, 2004.
30. Saul A. Kripke. Semantical considerations in modal logic. *Acta Philosophica Fenica*, 16:83–94, 1963.
31. Christopher Krügel, Engin Kirda, Darren Mutz, William K. Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, pages 207–226, 2005.
32. Pierre Luc Lespérance. Detecting variants of known attacks using temporal logic. In *WPTACT*, 2005.
33. Jacques Louis Lions. ARIANE 5: Flight 501 failure. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.
34. John McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, 2000.
35. James Newsome, Brad Karp, and Dawn Xiaodong Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.
36. Udo Payer and Stefan Kraxberger. Polymorphic code detection with GA optimized markov models. In *Communications and Multimedia Security*, pages 210–219, 2005.
37. Udo Payer, Peter Teufl, and Mario Lamberger. Hybrid engine for polymorphic shellcode detection. In Klaus Julisch and Christopher Krügel, editors, *DIMVA*, volume 3548 of *Lecture Notes in Computer Science*, pages 19–31. Springer, 2005.
38. Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.
39. Rix. Writing IA32 alphanumeric shellcodes. http://www.phrack.org/issues.html?issue=57&id=15.
40. Dragos Ruiu. Snort preprocessor - multi-architecture mutated NOP sled detector.
41. Matias Sedalo. JempiScode. http://goodfellas.shellcode.com.ar/proyectos.html.

42. Colin Stirling. Modal and temporal logics for processes. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency : structure versus automata*, pages 149–237. Springer, 1996.
43. Mehdi Talbi. IDS-logic. http://www.rennes.supelec.fr/ren/perso/mtalbi/outils/IDS-Logic.tar.gz.
44. Thomas Toth and Christopher Krügel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, pages 274–291, 2002.
45. Rafal Wojtczuk. The advanced return-into-lib(c) exploits: PAX case study. http://www.phrack.org/issues.html?issue=58&id=4.