J Comput Virol (2009) 5:247–261
DOI 10.1007/s11416-008-0095-z

EICAR 2008 EXTENDED VERSION

Functional polymorphic engines: formalisation, implementation and use cases

Grégoire Jacob · Eric Filiol · Hervé Debar

Received: 20 January 2008 / Revised: 19 June 2008 / Accepted: 27 June 2008 / Published online: 16 July 2008

Abstract With regards to the known shortcomings suffered by form-based detection, an increasing number of antivirus products considers behavioral detection. Following this trend, form-based mutations could become function-based with the apparition of functional polymorphism: a third generation of mutation mechanism, specially designed to address behavioral detection. In effect, a same global behavior or purpose (replication, propagation, residency, etc.) can be achieved through different functional solutions, thus leaving space for possible mutations. Whereas actual form-based mutation techniques mainly modify the code structure of malware, functional mutations modify the code functionality, and more particularly the resulting interaction scheme with the operating system and other software. These functional mutations could not be achieved without reaching a semantic level of interpretation, higher than actual techniques remaining purely syntactic. Drawing a parallel, this article underlines the consequent relation existing between functional polymorphic engines and compilers. By studying the associated mutation properties, we prove that these engines exhibit logarithmic entropy and result in a NP-complete complexity for behavioral detection. The implementation of a prototype is finally addressed as well as its possible use for antivirus testing and software protection.

G. Jacob (✉) · E. Filiol
French Army Signal Academy, Virology and Cryptology Lab., Rennes, France
e-mail: gregoire.jacob@gmail.com

E. Filiol
e-mail: eric.filiol@esat.terre.defense.gouv.fr

G. Jacob · H. Debar
France Télécom R&D, Caen, France
e-mail: gregoire.jacob@orange-ftgroup.com

H. Debar
e-mail: herve.debar@orange-ftgroup.com

1 Introduction

It is commonly acknowledged that form-based detection relying on byte signatures is eventually vowed to fail. As a consequence, malware researchers are considering new generations of detection techniques and in particular behavioral detection which can be deployed dynamically [1]. Unfortunately, for each detection solution put forward, the attackers have developed dedicated counter-measures. Similarly, functional polymorphism could be the third generation of mutation mechanism, following polymorphism and metamorphism, specifically designed to address behavioral detection. In effect, behavioral detection relies on the identification of malicious functionalities exhibited by malware (replication, propagation, residency, etc.). Each one of these functionalities can be implemented through different technical solutions leaving some degrees of freedom for possible functional mutations without undermining the originally intended purpose.

Up until now, such functional modifications have already been used by malware writers to avoid detection: modification, substitution, addition or removing of functionality blocks are common practices. The numerous versions of the Bagle e-mail worm, referenced by the different observatories, are typical examples of simple functional modifications (modifying mail subject, new backdoor, adding peer-to-peer sharing) [2]. Even if malware writers do not start their work from scratch, the generation of new variants from an original strain mostly remains manually achieved. However, some attempts of automation are already under way with the development of virus construction kits. Different engines can be cited from which the most popular ones are probably the Virus Construction Lab (VCL), the Phalcon/Skism Mass-Produced Code Generator (PS-MPC) and the Phalcon/Skism's G2 Virus Generator (G2) [3,4]. Still, the supplied customization options remain quite basic at the functional level (choice between appending, overwriting or companion infection, choice between encryption or plain code). Between two variants generated according to similar options, the differentiation is in fact achieved through metamorphic modifications and no real functional variation is deployed for a given functionality.

In some ways, previous works on mimicry attacks, led in host-based intrusion detection, seem more related to functional mutations [5,6]. The principle of mimicry attacks is to forge payloads containing a complete fixed attack hidden within a sequence of system calls imitating a legitimate application. By imitation, these forged payloads can bypass anomaly-based detectors while keeping the same effect on the system than the original attack. However, with regards to malware detection, most behavioral models are based on malicious signatures similar to those used by misuse-based intrusion detectors. Our approach will thus be slightly different from mimicry attacks: instead of including interleaved blank operations inside our code, the functional mutations we have designed enumerate the possible solutions to achieve a malicious behavior.

From these observations, and because anticipation is a key point in the antiviral struggle, we try to foresee and study the possible future threat that automated functional mutations could represent. The article is articulated according to the following structure. A brief overview is first drawn up upon the existing syntactic mutation mechanisms (Sect. 2). This first overview additionally highlights the necessary enhancements in order to reach mutations at the functional level. The following part is dedicated to formalization: functional mutations are introduced using compiler theory (Sect. 3). A resulting mutation entropy and detection complexity are then deduced from the formalism. The rest of the paper establishes a bridge between formalization and implementation (Sect. 4) and explores different use cases in antivirus assessment and software protection (Sects. 5 and 6).

2 From form-based to functional mutations

At the present time, polymorphism and metamorphism constitute the two major advances in code mutation. These two mutation mechanisms modify the assembly code at a syntactic level in order to conceal any similarity between two mutated variants. Considering the most advanced techniques in metamorphism, embedded in engines such as MetaPHOR [7], they remain based on practical obfuscation operations. These operations either directly modify the instructions (register reassignment, substitution of equivalent instructions enabled by translation into an intermediate pseudo-language) or globally modify the code structure and its possible execution paths (junk code insertion, instruction permutations, introduction of opaque predicates) [8, p. 148, 9, p. 269, 10]. Filiol, in a recent article, formalized the set of metamorphic transformations as rewriting rules from an original grammar describing the malware syntax, to a second mutated form [11]. He actually proved that well-chosen metamorphism rules could lead to the undecidability of the detection of the mutated forms, whereas detection remains NP-complete for polymorphic malware [12]. In practice, the substitution of equivalent instructions is undoubtfully the metamorphic technique which proves the most difficult to thwart for actual detectors [13]. Sequences of equivalent instructions may have different purposes but their combined execution have the same global effect on the memory. The main reason of their detection complexity is due to the fact that they do not only alter the program syntax but, to a lesser extent, also its semantic.

Nevertheless, even the substitution of equivalent instructions does not modify a priori the use made of the system services and resources (these accesses will be denoted by the terms "interaction scheme" within the paper). Using behavioral detection, the mutated variants should theoretically remain detected because of their identical interaction schemes. To overcome the simple instruction level of the existing mutation techniques, the next real challenge in code mutation lies in the research of different functionalities (computations and interactions) achieving the same purpose. To express an equivalence in terms of purpose, the manipulations must necessarily be performed at a semantic level working on more complex structures than simple instructions. Basically, a functionality is the combination of basic instructions but also different system calls and parameters. Two functionalities can be said equivalent if their executions impact similarly the behavior of the host system and no longer, if they simply exhibit the same effect on memory. For example, under a Windows operating system, modifying a run registry key or the system.ini file have different effects on memory but eventually the same consequence, that is to say, to automatically start a program during the boot session. According to this guideline, we had already introduced briefly the concept of functional mutations in a previous article [14]. We now want to provide a solid formalization and give a proof of automated feasibility.

3 Compiler theory applied to polymorphism

Basically, the purpose of a functional polymorphic engine is to translate the final purpose of a behavior into executable code. This behavior description is often conveyed by a specifically designed language, guaranteeing that every mutated form will consistently perform the intended task. Consequently, the engine functioning is similar to the one of a compiler. Yet, the peculiarity of this engine is that several successive executions must result in strongly different variants, thus introducing the concept of non-deterministic compiler. In effect, to avoid behavioral detection, the malware must modify their functionalities and interaction schemes at each execution. Before going any deeper in the formalisation, we think that it is important to remind briefly some important definitions, in particular to explain the notations that will be used along the article. Some of them can be found in reference books about grammars and automaton [15] or in the literature about attribute grammars [16, 17, Lect. 15, p. 14].

Definition 1 A context-free grammar G is a quadruplet $<V, \Sigma, S, P>$ where:

- V is the finite set of non-terminal symbols also called variables,
- Σ is the finite set or alphabet of terminal symbols forming the language,
- $S \in V$ is the start symbol,
- P is the set of production rules of the form $V \rightarrow \{V \cup \Sigma\}^*$.

Definition 2 An attribute grammar G_A is a triplet $<G, D, E>$ where:

- G is originally a context-free grammar $<V, \Sigma, S, P>$,
- let $Att = Syn \uplus Inh$ be a set of attributes divided between the synthesized and the inherited attributes, and $D = \cup_{\alpha \in Att} D_\alpha$ be the union of their sets of values,
- let $att : X \in \{V \cup \Sigma\} \longrightarrow att(X) \in Att^*$ be an attribute assignment function,
- every production rule $\pi \in P$ of the form $Y_0, \longrightarrow, Y_1, \ldots, Y_n$ determines a set of attributes $Var_\pi = \cup_{i \in \{0,\ldots,n\}} \{Y_i.\alpha \mid \alpha \in att(Y_i)\}$ partitioned between inner variables: $In_\pi = \{Y_0.\alpha | \alpha \in att(Y_0) \cap Syn\} \cup \{Y_i.\alpha | i \neq 0, \alpha \in att(Y_i) \cap Inh\}$,
  and outer variables: $Out_\pi = Var_\pi \backslash In_\pi$,
- E is a set of semantic rules such as for any production rule $\pi \in P$, for each inner variable $Y_i.\alpha \in In_\pi$, there is exactly one rule of the form $Y_i.\alpha = f(Y_1.\alpha_1, \ldots, Y_n.\alpha_n)$ where $Y_j.\alpha_k \in Out_\pi$ and $f\colon D_{\alpha_1} \times \cdots \times D_{\alpha_n} \rightarrow D_\alpha$.

Context-free grammars can basically be evaluated by pushdown automata. In compilation, these automata are used for building the derivation tree according to the syntax of the source. In the case of attribute grammars, a pushdown automaton is still mandatory to parse the syntax but an additional attribute evaluator is required to evaluate the associated semantic rules. The attribute evaluation may be solved by two kinds of methods: topological sorting or recursive functions [17, lect.18 pp. 3]. In this article, we will only consider the topological sorting approach whose description is given just after the definition of a pushdown automaton.

Definition 3 A pushdown automaton A is a seven-tuple $<Q, \Sigma, \Gamma, \delta, q_0, Z_0, F>$ where:

- Q is the finite set of states,
- Σ is the alphabet of input symbols,
- Γ is the alphabet of stack symbols,
- δ is the transition function of the form $Q \times \{\Sigma \cup \epsilon\} \times \Gamma \rightarrow Q \times \{\Gamma \cup \epsilon\}$,
- $q_0 \in Q$ is the initial state,
- $Z_0 \in \Gamma$ is the initial symbol on the stack,
- $F \subset Q$ is the set of accepting states.

Definition 4 Algorithm for the attribute evaluation by topological sorting:

- **Input:** an attributed grammar G_A, a simple derivation tree T of G_A, and an initial valuation for the terminal symbols $v : Syn_\Sigma \rightarrow D$. Let Var_T be the set of attributes of T and E_T be its attribute equation system.
- **Procedure:**

```
I.  Let $Var := Var_T \backslash Syn_\Sigma$.
II. While($Var \neq \emptyset$) do
    1. Choose $x \in Var$ such as $x = f(x_1, \ldots, x_n)$
       $\in E_t$ and $\forall i, x_i \notin Var$.
    2. $v(x) := f(v(x_1), \ldots, v(x_n))$.
    3. $Var := Var \backslash \{x\}$.
```

- **Output:** Solution $v : Var_T \rightarrow V$.

We have now sufficient concepts to introduce a simplified definition of a compiler as a basis for our work. We will use the most uncluttered vision of a compiler without the intervention of intermediate languages or optimization, leaving only two steps: verification building the attributed derivation tree and translation generating the executable code as shown in Fig. 1.

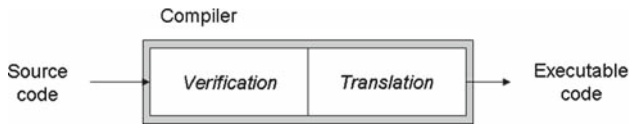


Fig. 1 Generic view of a simplified compiler. This is the simplest decomposition of a compiler. Lexical analysis, the use of intermediate languages and optimization techniques have been willingly ignored for the sake of simplicity

Definition 5 A compiler C is a is a quintuplet $<G_S, I, A_{G_S}, V_{G_S}, R_T>$ where:

- G_S $=<G, D, E>$ is the attribute grammar of the source code, based on the context-free grammar $G =<V, \Sigma, S, P>$,
- I is the alphabet of instructions describing the targeted machine,
- A_{G_S} is the pushdown automaton used in the verification process accepting the syntax of the source grammar G_S and producing the derivation tree T,
- V_{G_S} is the attribute evaluator based on topological sorting used as a second step during the verification of the derivation tree T,
- $R_T \subseteq \{(\Sigma \times D)^* \times I^*\}$ is a rewriting system (also called semi-Thue system) translating the nodes of the form $(\Sigma \times D)$ from the attributed derivation tree into executable code over the instruction set I.

3.1 Functional polymorphism formalization

The required background about compiler theory being introduced, we can now move to the new formalism. It is important to keep in mind that functional metamorphism works at a semantic level, just like compilers do. The final purpose of each behavior, in other words its semantic interpretation, must be expressed in a attribute grammar. An example is addressed in the next part but right now the formalization should be independent from the considered grammar. A behavior can then be implemented in several ways corresponding to the different possible semantically attributed derivation trees.

The mutation approach will thus be slightly different from compilation. A compiler, given a source code ω in input verifies first its syntax. The automaton A_{G_S} will accept the source code if and only if $\hat{\delta}(q_0, \omega) \in F$. Given an initial attribute valuation for terminals v, the evaluator V_{G_S} of the compiler tries to build a complete valuation satisfying the equation system. In case of success, the source code is then translated according to the rewriting system R_T: $(\omega, v) \overset{*}{\Longrightarrow}_{R_T} \omega'$ with $\omega' \in I^*$. Whereas, the purpose of the mutation engine is to keep the original functionality through divers instantiation. It will thus take in input a start symbol S from the behavior grammar G. Instead of verification, the engine achieves a derivation of the grammar: $S \overset{*}{\rightarrow}_{G_S} \omega$ with $\omega \in T^*$. In a second step, this derivation tree is attributed by generation of a new valuation satisfying the equation system of G_S. The rest of the translation process is then identical. The main idea is illustrated in Fig. 2. No additional verification is required since by automated construction, the code is obviously syntactically and semantically correct.

During derivation several derivation trees may syntactically be possible. Derivation will thus be embedded in a

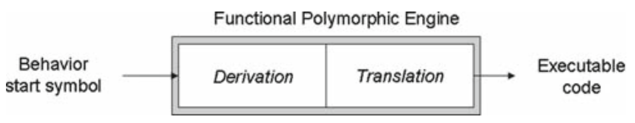
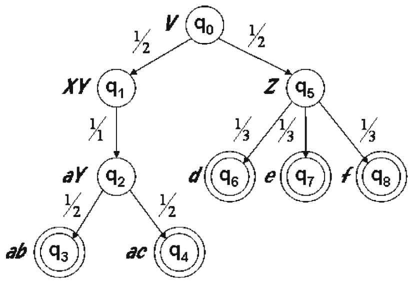


Fig. 2 Generic view of a functional polymorphic engine. With regards to the generic compiler, the main difference lies in the substitution of the verification process by a derivation process

probabilistic automaton that will replace the deterministic one used for verification. For a short example, let us define the following grammar (on the left) and its associated derivation probabilistic automaton (on the right):

$$\begin{aligned} <V> &\rightarrow XY \mid Z \\ <X> &\rightarrow a \\ <Y> &\rightarrow b \mid c \\ <Z> &\rightarrow d \mid e \mid f \end{aligned}$$



With regards to semantic verification, the equation system for attribute evaluation can hardly be modified without loosing the grammar coherence. However, the initial valuation for the terminals of the grammar leaves some degrees of freedom (see Inputs in Definition 4). Several initial valuations may satisfy the system of equations and the engine can randomly chose between them. In Definition 4, a possible algorithm is given for attribute generation. Based on a recursive procedure, the algorithm explores the space of possible solutions as a decision tree with backtracking facilities.

Definition 6 Recursive algorithm for the attribute generation:

- **Input:** an attributed grammar G_A, a simple derivation tree T of G_A and an empty initial valuation v. Let Var_T be the set of attributes of T and E_T be its attribute equation system.
- **Procedure: recursive_generation**

```
I.   If $Var_T = \emptyset$ do return true.
II.  Choose $x \in Var_T$ such as $x$ has the minimum
     dependency i.e. the minimal number of semantic
     rules: min(card($\{\pi | x \in Var_\pi\}$)).
III. $Var_T := Var_T \backslash \{x\}$.
IV.  Solve the sub-system $E_{T_x}$ of equations from $E_T$
     containing $x$, $x$ is then reduced to a solution domain
     $D_s \subset D_x$.
V.   While($D_s \neq \emptyset$) do
     1. Choose randomly $s \in D_s$.
     2. $v(x) := s$.
     3. Call recursive_generation.
     4. If false is returned do
        $D_s := D_s \backslash s$.
        5. Else do
                Return true.
    VI. $Var_T := Var_T \cup \{x\}$.
    VII. Return false.
```

– **Output:** (Only if true is returned, otherwise no possible valuation)
Solution $v : Var_T \to V$.

This algorithm solves independently the different subsystems of equations following the increasing dependency (increasing number of semantic rules for a given attribute). At each step, the set of possible values for this attribute is reduced. If such a decision makes successive steps unfruitful, the algorithm allows backtracking and explores a new branch. In fact, the main drawback of this algorithm is that it basically proceeds by brute-force. Some additional optimizations could surely be found in addition to the choice of minimum dependency. However, in the present case, the probability of success in reasonable time is quite high. First, because the set of values for attributes are bound spaces of discrete values. Then, because the semantic equations often remain quite basic: linear equations most of the time.

Generally speaking, attribute generation is critical since attribute values are used for selecting the right rewriting rule between the different associated to a same terminal. This finally leads us to the following definition for a functional polymorphic engine:

Definition 7 A functional polymorphic engine M is a quintuplet $<G_S, I, A_{G_S}, V_{G_S}, R_T>$ where:

– $G_S =<G, D, E>$ is the attribute grammar of the source code, based on the context-free grammar $G =<V, \Sigma, S, P>$,
– I is the alphabet of instructions describing the targeted machine,
– A_{G_S} is the probabilistic finite automaton deriving the start symbol S into simple syntactic derivation tree T according to G_S,
– V_{G_S} is the attribute generator determining a random initial valuation for the terminals satisfying the equation system of T,
– $R_T \subseteq \{(\Sigma \times D)^* \times I^*\}$ is a rewriting system (also called semi-Thue system) translating the nodes of the form $(\Sigma \times D)$ from the attributed derivation tree into executable code over the instruction set I.

3.2 Characteristics of the mutation

3.2.1 Mutation entropy

The information entropy introduced by C.E. Shannon makes it possible to measure the uncertainty associated with the mutation process which is particularly interesting to assess the engine effectiveness [18]. The mutation engine can be modeled as a communication channel receiving data from a source: the original description file from the hard drive, and transmitting it to a recipient: the final executable built in the process memory. During the transmission, some noise is introduced by the engine through the mutations.

We have based our reasoning on an average case requiring the definition of specific parameters:

– The average depth d of a grammar which is the average number of production rules to apply during derivation in order to reach the final word. It is equivalent to the average number of intermediate states required by the automaton before to reach an accepting one.
– The average number n of alternative options for a production rule. It is equivalent to the average number of successors for a given state of the automaton.
– The average number s of possible initial valuations given a derivation tree T. It is possible to bound this value considering the best and worst cases. With regards to entropy, the worst case is reached when the attribute equation system accepts a single initial valuation as solution. On the other hand, the best case is reached when all the attributes of the terminal symbols from the tree T are independent. Using the notations from the definitions, then the initial value of an attribute $\alpha \in syn_\Sigma$ can be any value from the domain D_α. This can be summed up by the following inequality: $1 \le s \le \Pi_{\alpha \in Var_T \cap Syn_\Sigma} card(D_\alpha)$.

There are two points over the channel where some uncertainty is created: the random derivation and the choice of the attribute valuation. This leads us to the following proposition:

Proposition 1 *By considering uniformly distributed random choices, the average entropy is given by:* $H(mutation) = d \times \log_2(n) + \log_2(s)$.

Proof Let us begin by calculating the probability associated to the syntactic derivation of a word ω which is obtained by the path of state $\pi_\omega = e_1, \ldots, e_d$. Considering a probabilistic automaton, the probability of selecting a given state among the possible successors is only dependent of the current one like in a first-order Markovian process:

$$P(\omega) = P(e_0)\Pi_{i=1}^{d} P(e_i|e_{i-1})$$

The starting state e_0 is mandatory which gives us:

$$P(e_0) = 1$$

By reasoning on an average basis, we know that for any ω derived from G, d states are reached. At each step, n options are available:

$$P(e_{i+1}|e_i) = \frac{1}{n}$$
$$P(\omega) = \left(\frac{1}{n}\right)^d$$

Given the derivation ω, the engine chose randomly a possible initial valuation v with equivalent probability

$$P(v|\omega) = \frac{1}{s}$$

Which leads us to this result:

$$P(\omega, v) = P(\omega)P(v|\omega) = \frac{1}{sn^d}$$

By a similar reasoning we can calculate the average number of possible attributed derivation trees:

$$\text{card}(L(G)) = sn^d$$

The entropy of the derivation is thus given by:

$$\begin{aligned} H(\text{mutation}) &= -\Sigma_{(\omega,v)\in L(G)} P(\omega, v)\log_2(P(\omega, v)) \\ &= -\text{card}(L(G))P(\omega, v)\log_2(P(\omega, v)) \\ &= -sn^d\left(\frac{1}{sn^d}\right)\left(\log_2\left(\left(\frac{1}{sn^d}\right)\right)\right) \\ &= d(\log_2(n)) + \log_2(s) \end{aligned}$$

□

This result is based on specific hypothesises but it gives, if not precise, a pertinent assessment of the mutation effectiveness. It may be interesting to interpret it. In fact, d and n are syntactic factors settled by the behavior grammar. This grammar is designed to convey the minimal expression of the final functionality with the best coverage. Consequently, it cannot be the subject of easy modifications. The semantic factor s remains the main degree of liberty and enables a logarithmic increase of the mutation entropy. Several semantic parameters can impact the value of s: the number of attributes for each symbol, the range of their possible values, and the number of dependencies between them. This statement is quite important since the number of equivalent rewriting rules for a terminal symbol is directly proportional to the possible values taken by its attributes. This underlines the fact that functional polymorphism goes beyond the simple syntactic level.

3.2.2 Detection complexity

If mutation entropy was interesting from the perspective of the attacker, detection complexity proves more relevant from the defender's perspective. Let us now focus on the complexity of the behavioral detection problem for functional polymorphic malware of finite size. Considering actual behavioral detectors, most of them rely on predefined behavior signatures. According to previous works, these signatures may be expressed as Boolean formula [14,19]. Behavioral detectors can be divided into two classes: dynamic simulation-based detectors relying on sequences of observable events (system call traces) and static formal verifier relying on instruction meta-structures (graphs, temporal logic formula) [1]. Considering an observable event i (resp. an instruction) and a position j in the sequence (resp. the structure), let us define a Boolean variable:

$$X_{i,j} = \begin{cases} 1 & \text{if } i \text{ is present at the position } j \\ 0 & \text{otherwise} \end{cases}$$

These Boolean variables are combined into a formula representing the whole sequence or meta-structure. Some events (resp. instructions) may be interchangeable at a given position. A sequence (resp. a structure) is then a Boolean formula in its conjunctive normal form (CNF):

$$X_{s_k} = X_{i_1,1} \wedge (X_{i_2,2} \vee X_{i'_2,2} \vee \cdots) \wedge \cdots \wedge (X_{i_n,n} \vee \ldots)$$

A given behavior can finally be instantiated through various equivalent sequences (resp. structures). A behavior β is thus modelled by a disjunction of formulae:

$$X_\beta = X_{s_1} \wedge X_{s_2} \wedge \cdots \wedge X_{s_n}$$

The global behavioral detection scheme is given by a Boolean correlation function ϕ_c over the v different behaviors referenced in the database:

$$\beta_M = \phi_c(X_{\beta_1}, \ldots, X_{\beta_v})$$

According to this modelling, following an similar reasoning approach to the one used by D. Spinellis to study the impact of syntactic polymorphism on signature detection [12], we can likewise reduce the behavioral detection problem to a satisfiability problem:

Proposition 2 *The behavioral detection of functional polymorphic malware with finite size is NP-complete.*

Proof Let D be a behavioral detector and let us assume that it can reliably determine in P-time whether a program exhibit or not a mutated form of a given behavior B. We will now use D for determining the satisfiability of N-terms Boolean formulae.

We know from this section that a behavior can be described by its signature: a Boolean formula S. Using S, we can create a virus archetype A as a triple (f, s, c) where:

–c is an evolving integer used to generate a new interpretation of the formula S,

–s is a Boolean value indicating if S has been satisfied,

–f is the duplication function. f simulates the functional polymorphism by computing a new value for c, which indirectly creates a new interpretation for S. f finally updates s using the new interpretation.

For D to detect whether a given virus is a mutated version of a known functional polymorphic strain, it must then determine

Fig. 3 Duplication description. Basically, duplication consists in copying the code from the running virus ($this$) into a permanent object newly created (obj_perm). This grammar is an extended version of the one introduced in the previous article [21]

$$\begin{array}{lll}
(i) & <Duplication> & ::= <Creation><Opening><Reading><Writing> \\
 & & \;\;|\; <Opening><Reading><Creation><Writing> \\
 & & \;\;|\; <Opening><Creation><InterleavedRW> \\
 & & \;\;|\; <Creation><Opening><InterleavedRW> \\
 & & \;\;|\; <Opening><DirectTransfer> \\
(ii) & <Creation> & ::= create \quad obj\_perm; \\
(iii) & <Opening> & ::= open \quad this; \\
(iv) & <Reading> & ::= receive \quad var \leftarrow this; \\
(v) & <Writing> & ::= send \quad var \rightarrow obj\_perm; \\
(vi) & <InterleavedRW> & ::= while(receive \quad var \leftarrow this;)then\{ \\
 & & \qquad send \quad var \rightarrow obj\_perm; \\
 & & \quad \} \\
(vii) & <DirectTransfer> & ::= send \quad this \rightarrow obj\_perm;
\end{array}$$

if S will ever be satisfied. In the case of functional polymorphic engines, we have seen that S is a disjunction of CNF formulae modelling the sequences of events or meta-structures of instructions. The detector D should then be able to solve the SAT problem for at least one clause of the disjunction. By reduction, the detection problem is equivalent to solving the SAT problem which has been proven NP-complete [20]. $\square$

4 Implementation aspects

Like we have already stated, any attempt of semantic manipulation requires a high-level description language conveyed by a grammar. This language expresses an equivalence in terms of purpose between two functionalities deriving from a same start symbol, meaning that every mutated form will consistently perform the intended task. In the context of this paper, we have chosen to use the grammars introduced in a previous article to model the main malicious behaviors through their final purpose [21]. The adopted perspective is object-oriented: the malware embed internal mechanisms and attributes but also provide external interfaces for interaction with adversaries. These adversaries are classified according to their use in malware's lifecycle: auto-reference, permanent objects, communicating objects or boot objects. A behavior description can be seen in Fig. 3, as an example. Anyhow, the same reasoning could be applied without loss of generosity, to any other behavioral model possibly described by a language: general behavior patterns from VIDES [22], high-level actions from GateKeeper [23], etc. Let us now introduce how a functional mutation engine can be built in practice.

4.1 Prototype architecture and results

As stated in the formalization, the functional polymorphism engine is divided between two components respectively responsible for the derivation and the translation. Each of these components is then divided between different modules briefly described below. The overall architecture of the prototype and the junction of the different modules is illustrated in Fig. 4.

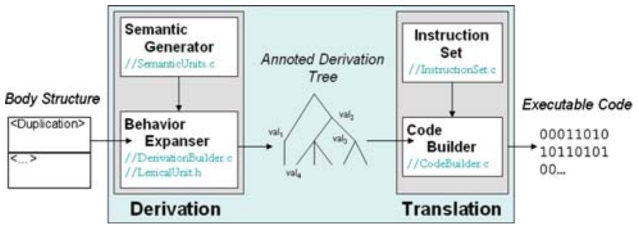


Fig. 4 Architecture of the prototype. This schematic description of the architecture reveals the junctions between the different modules of the prototype

Behavior expanser: The behavior expanser is part of the derivation component. This module embeds the syntactic rules of the behavior language inside a probabilistic automaton in order to build a random derivation tree. During derivation, it calls on the semantic generator services to annotate the tree.

Semantic generator: This generator is responsible for creating the semantic attributes associated to the different production rules. It embeds the semantic equations to guarantee the coherence of the valuation.

Code builder: The code builder is the entry point of the translation component. This module reads the derivation tree and its semantic annotations in order to build the corresponding executable code. It uses the basic building blocks supplied by the instruction set in order to build the malware body and updates these blocks according to the semantic attributes.

Instruction set: The instruction set defines the meta-structures of instructions corresponding to the basic operations: arithmetic ones for example but also more complex operations like the parameter passing of system calls.

The prototype has been implemented in C and the basic building blocks are directly written in assembly. It is now operational and currently supports four different behaviors used

```
GetModuleFileName         fopen          GetModuleFileName
 "kernel32.dll"       "msvcrt.dll"        "kernel32.dll"
   CreateFile       GetModuleFileName        CopyFile
 "kernel32.dll"      "kernel32.dll"       "kernel32.dll"
   CreateFile             fopen
 "kernel32.dll"       "msvcrt.dll"
    ReadFile              fseek
 "kernel32.dll"       "msvcrt.dll"
   WriteFile              ftell
 "kernel32.dll"       "msvcrt.dll"
    ReadFile             frewind
 "kernel32.dll"       "msvcrt.dll"
   WriteFile              malloc
 "kernel32.dll"       "msvcrt.dll"
    ReadFile              fread
 "kernel32.dll"       "msvcrt.dll"
      ...                 fwrite
      ...             "msvcrt.dll"
      ...
    ReadFile
 "kernel32.dll"
   WriteFile
 "kernel32.dll"
```

Fig. 5 Execution traces. This figure collects the different dll function calls made over three consecutive executions. These are just extracts relative to the duplication behavior (interleaved read/write on the left, one block reading and writing in the middle, direct transfer on the right). This is typically the kind of information collected by an antivirus product for behavioral detection

in P2P/Mail worms: duplication, propagation, residency and overinfection test. The global size of the code is about 40 KB and uses less than 30 basic building blocks (from 4 to 80 bytes in size). From this, the engine is able to build thousands of basic derivations only by modifying the syntax and the types of the semantic objects (registry key, file, socket, etc.), and even more if we consider the differences in terms of object location or attributes as we will see a little bit further. To give you an hint of the result, Fig. 5 gathers three traces relative to duplication in three consecutive executions of the engine.

Going back to the formalization part, there are basically two degrees of liberty in the mutation. One lies in the different possible derivations from a start symbol. The other one lies in the generation of semantic attributes that will determine the rewriting rule to use. The behavior expanser and the semantic generator are thus the real core of the engine more than the code builder itself. The two next parts makes explicit their technical details.

4.2 Syntactic behavior expansion

The first level of mutation is achieved by a random derivation of the grammar performed by the behavior expanser which replaces the usual parser used for verification. The structure of its source code is quite similar to the one of a grammar parser. However, instead of choosing the following production rules according to the current symbol under the parsing head, the expanser unrolls the production rules choosing randomly between the different options at each step. From a start symbol, the expanser generates a valid derivation tree inside the possibility space. A simplified sample from the source code is shown in Fig. 6 in direct relation with the grammar given in Fig. 3. Notice that the non-determinism of the derivation automaton does not lift the deterministic constraints on the grammar, it still requires to be LL(k) or LR(k) to build the executable code.

```
int DuplicationExpand(...){
    int uiWhich=RandomGenerator(5);
    switch(uiWhich){
    case 1:
        CreationExpand(...);
        OpeningExpand(...);
        ReadingExpand(...);
        WritingExpand(...);
        break;
    case 2:
        OpeningExpand(...);
        ...
    ...
    case 5:
        OpeningExpand(...);
        TransferExpand(...);
        break;
    }
}
```

Fig. 6 Derivation functions. This short code sample illustrates the inclusive call sequences. Contrary to common parsers, the following step is not determined by the current syntactic unit but randomly chosen

```
<Overinfection>
    <marker= "marker_name"\ >
< \ Overinfection>
<Duplication>
    <target= "target_name"\ >
< \ Duplication>
<Residency>
< \ Residency>
<Propagation>
    <carrier= "lure_name"\ >
< \ Propagation>
<Payload>
< \ Payload>
```

Fig. 7 Global structure of a P2P/mail worm. This file, written in a format similar to XML, describes the global structure of the worm. Using dedicated tags, it is possible to specify certain parameters of the behaviors, such as the name of the duplicated instance for example

In input, the derivation process must be fed with a global description of the malware. The purpose of this description is to determine the articulation of the different behaviors inside the malware body. The start symbols of the behavior grammars are used as basic blocks to build a description in a format similar to XML. An example of a generic P2P/mail worm is provided in Fig. 7 with additional tags to customize the behaviors. The resulting output from the expanser will be a syntactic derivation tree satisfying the behavior grammar.

4.3 Code generation according to semantic

The second level of mutation is achieved by the semantic generator through the generation of semantic attributes satis-

Fig. 8 Duplication attributed description. Semantic rules have been added for the binding of variables and objects as well as typing

```
(i)   <Duplication>      ::= <Creation><Opening><Reading><Writing>
                           | <Opening><Reading><Creation><Writing>
                           | <Opening><Creation><InterleavedRW>
                           | <Creation><Opening><InterleavedRW>
                           | <Opening><DirectTransfer>
 {      <Writing>.objId=<Creation>.objId
        <Writing>.objType=<Creation>.objType
        <Writing>.varId=<Reading>.varId
        <InterleavedRW>.objId=<Creation>.objId
        <InterleavedRW>.objType=<Creation>.objType   }
(ii)  <Creation>         ::= create   obj_perm;
 {      <Creation>.objId=obj_perm.objId
        <Creation>.objType=obj_perm.objType   }
(iii) <Opening>          ::= open   this;
(iv)  <Reading>          ::= receive   var ← this;
 {      <Reading>.varId=var.varId   }
(v)   <Writing>          ::= send   var → obj_perm;
 {      <Writing>.varId=var.varId
        <Writing>.objId=obj_perm.objId
        <Writing>.objType=obj_perm.objType   }
(vi)  <InterleavedRW>    ::= while(receive   var ← this;)then{
                                 send   var → obj_perm;
                             }
 {      var_1.varId=var_2.varId
        <InterleavedRW>.objId=obj_perm.objId
        <InterleavedRW>.objType=obj_perm.objType   }
(vii) <DirectTransfer>   ::= send   this → obj_perm;
```

fying the attribute equation system. These annotations are particularly important since they will determined the rewriting rules to use for a given terminal symbol. The example from Fig. 3 has been rewritten using attribute equations in Fig. 8. These attribute equations are used for several purposes:

Object binding: The first step of the attribute evaluation is performed by binding the semantic objects. This mechanism identifies the different instances of objects and variables and guarantees they are coherently used. Object binding is basically used to constrain the data flow between objects. Let us consider the duplication example of Fig. 8. Object binding is done by affecting an attribute identifier ($objId$) to the permanent object and verifying that it is this same object that we open and then write to. This binding is not subject to mutation since it is constrained by our behavior grammar.

Object typing: The second step in the annotation process is performed by associating types to the different objects. In fact, it is type information that determines the rewriting rule to use. It is easy to understand that we dispose of several primitives to traduce a given grammar unit. If we take for example the command *create obj_perm*, it can be performed by several system calls depending on whether the object is a file or a registry key. It can easily be seen that object typing impact greatly the interaction scheme. By affecting a type to an object ($objType$), we reduce the set of possible translation rules to a singleton. In our polymorphic context, this affectation must be performed randomly between a range of coherent values.

Object characterization: This last step, absent in simple compilation, has been specifically added. Characterization randomly affects additional characteristics to object. These characteristics stored in object structures like the one described in Fig. 9 are then used as parameters for the built instructions:

- Access characterization which constrain the stream flow: unilateral or bilateral. It is particularly important in cases like the autoreference since running programs can only be accessed in reading mode.
- Localisation which determines the location of objects. It can be a simple path for a file or a subtree for registry keys.

```
struct obj_entry{
      unsigned long ulIdentifier;
      unsigned long pObjectHandle;
      char pcName[MAX_PATH];
      char pcLocation[MAX_PATH];
      unsigned int uiType;
      char pcAccess[4];
      unsigned long ulAttribute;
};
```

Fig. 9 Object semantic structure. This structure is used in the prototype to store the different semantic annotation generated to build the executable code

– Attributes which define basic properties of the object. Once again, a file, for example, can be hidden, compressed, ciphered or associated to the system according to the facilities offered by the file system.

When launching the application, after performing the derivation, the generated code is built in a newly allocated memory space with execution rights. Building dynamically the code introduces addressing problem to replace the linking process. In order to build relocatable code, all variables and objects as well as the import table are managed by the code builder in structures similar to the one in Fig. 9. Consequently, the generated code is able to address them directly through their handle without localization problem.

5 Use case for antivirus products assessment

Assessing antivirus products is still an open problem and few works have been led on the subject. Up until now, most test procedures simply confront malware detectors to known strains thereby solely assessing the coverage of their signature database. However, finding a procedure to assess the detection of unknown malware is far more complex. Fortunately, functional mutations could be used in the context of blackbox tests to address this issue and more particularly to assess the coverage of behavioral detection engines. A first methodology had been introduced in a previous article based on the manual simulation of functional modifications [14]. The idea was similar: achieve the same behavior through different instantiations. Unfortunately, the process was not wholly automated, requiring the manual generation of the new variants which finally proved quite prohibiting. The definition of an autonomous engine for functional mutations has allowed us to revise the process to make it fully automated.

5.1 Methodology

Typically, functional polymorphism engines convey a generic semantic model and translate it towards a specific instantiation (refinement procedure). Within the perspective of detection, this principle is reversed: the detector analyzes a given instantiation, interprets it, and compares it to a generic model. Unfortunately, severe problems of completeness and accuracy are often observed. By adopting the attacker's point of view, it proves easier to automatically enumerate significant variants of an original strain. As a consequence, functional polymorphic engines may be valuable tools to assess the coverage of behavioral detectors just like simple metamorphic mutations can be used for assessing the resilience of signature-based detection [24].

One could object that it may be very easy to establish a signature for the invariant core of our engine. However, this engine has not been developed to become an operational viable attack. This prototype has been implemented for research and testing purposes. Besides, the absence of signature is a prerequisite of the test procedure, otherwise form-based detection would hinder the evaluation by detecting preemtively the engine before any action of the behavioral detector.

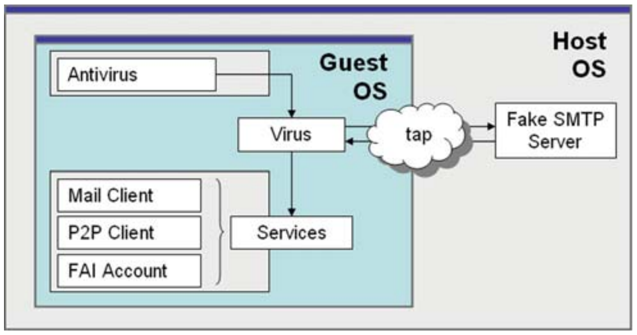


Fig. 10 Test platform. This figure pictures the different elements and services running on the platform, either on the host machine or inside the guest operating system

5.2 Test platform

The first step was to develop the prototype and, using on-demand scan, to make sure that no syntactic signature existed for it. A platform was then required to observe the execution of a piece of malware in an environment protected by the antivirus product to be tested. For this, we have chosen to use a virtual machine, mainly for two reasons: the first is to prevent any infection of the real machine to occur, and the second is the capability to reset the platform in a clean state in case the malware variants are not detected. The global architecture of the test platform is described in Fig. 10 and additional information are given below:

Guest machine: Qemu [25] has been used for the emulation of the virtual environment. Windows XP SP2 has then been installed and configured as a personal computer: additional services usually hijacked by malware have been installed such as a mail client and a peer-to-peer client. In addition, an ISP account has been configured with different account information like the associated SMTP server for example. Once the installation achieved, the disk image has been duplicated into clean copies, to receive the different antivirus products and the virus itself (without running it yet). From there, the tests simply consist in executing several times the engine in the virtual machine, the machine running in snapshot mode to restart it after each infection.

Host machine: A tap has been installed between the host machine and the guest machine in order to establish a virtual network communication between them. In parallel of the guest machine, a fake SMTP server was running on the host, listening on port 25, dumping the received SMTP packets and responding with the correct acknowledgements. The host file of guest OS had been previously rewritten in order to route all the traffic of the different servers toward the tap.

5.3 Evaluation deployment

The test platform is fully operational and has been deployed to assess different antivirus products whose results are given below (Sects. 5.3.1 to 5.3.4). Four products have been selected, integrating different levels and techniques of behavioral detection (behavioral blockers, heuristic, state automata [1]). Please keep in mind that the results are not given for a survey of the antivirus market but only to validate our procedure.

5.3.1 DrWeb results

According to the results shown in Table 1, no monitoring of the actions taken by the malware must be done in this version of DrWeb. However the editor announced a few months ago, the release of a new engine in addition to traditional signature scan and heuristic analysis: Origins Tracing^{TM} which is specifically designed to detect unknown malware. No more information is given on its functioning, we can only assume it is not based on behavioral models because the behaviors embedded in our mutation engine are inspired from common malware and are thus basically well known by analysts. It is simply the way they are deployed and combined which differs. If behavioral detection was integrated, some instantiations of standard behaviors among the hundreds of executions should at least have been recognized.

5.3.2 NOD32 results

According to the results shown in Table 2, NOD32 seems to use heuristics for behavior monitoring as the labels of the detected variants suggest. Crossing the results, these variants are all detected through their attempts to replicate: the target of the duplication cause the detection as written down in the log. If we look closer at these variants, the only common point they share is that they derive from the $<DirectTransfer>$ rule from duplication (see Sect. 4.3, Fig. 8). This particular derivation is translated using the system call CopyFile in order to copy the malicious code. On the other hand, the other duplication attempts using the standard ReadFile and WriteFile primitives are not detected. This interpretation does not seem inconsistent with our result: on average 20% of the variants should be derived from the $<DirectTransfer>$ rules and 14% were detected in practice, independently from the location of the target. With a greater number of tests we should come closer to the theoretical probability but still the observed gap is not too consequential.

Table 1 Detection results for DrWeb. Software version: DrWeb(R) Virus-Finding Engine—drweb32.dll (4,44,0,09176)/SpIDer Guard Service—Spidernt.exe (4,44,4,09260)/SpIDer Mail (R) for Windows—spidermail.exe (4,44,1,12220)/Signature base: 14.01.2008/283790 entries

DrWeb Anti-virus for Windows 4.44 (2008); Editor: Doctor Web, Ltd.

Number of executions	Detection rate (%): resident protection	Detection rate (%): mail protection
500	0 (0%)	0 (0%)

Table 2 Detection results for NOD32: threat sense early warning system, protection from potentially unwanted application and resident protection activated; signature database: 2740(20071221)/Antivirus and Antispam scanner module: 1001(20071221)/Advanced heuristic module: 1068(20071119)

NOD32 Anti-virus 3.0.621.0 (2008); Editor: ESET

Number of executions	Detection rate (%): real-time file system protection
500	71 probably unknown new Heur_PE virus (14%)

5.3.3 Product A results [1]

Product A, whose results are given in Table 3, combines two different methods of behavioral detection: behavioral blocking for registry monitoring and global activity monitoring. Behavioral blocking is preemptive and thus the first engine to detect the different variants. The tests have resulted in 27% of detection which, after verification, covers all the variants making themselves resident through the run registry key. This detection rate is consistent with the probability of one in three to choose this method of residency. If all attempts have been detected, however, no correlation is done and the final decision is left to the user. To follow the process, we have by default accepted the operation in order to keep on with the detection.

The second detection pass relies on activity monitoring and seems independent from the behavioral blockers and its decisions. The monitoring engine correlates a certain number of actions (file creations, file or registry modifications, etc.) to support its decision. Two generic threats are detected but with a relatively low rate according to the results of Table 3: generic P2P Worms and generic Trojans with about 10%

[1] Product has been anonymized because the terms concerning blackbox evaluation in the licence contract were unclear. The product is not to be used in automatic, semi-automatic or manual tools designed to create virus signatures, or virus detectors.

Table 3 Detection results for Product A

Product A (2008); Editor: X

Number of executions		Non-labelled	Generic P2PWorm[a]	Generic Trojan[b]	Total
500	Blocking run registering	98 (19.6%)	11 (2.2%)	26 (5.2%)	135 (27%)
	Non-blocked	300 (60%)	42 (8.4%)	23 (4.6%)	365 (73%)
	Total	398 (79.6%)	53 (10.6%)	49 (9.8%)	500 (100%)

[a] Attempting to copy towards a network resource
[b] Registering its copy on the system

Table 4 Detection results for Product B

Product B (2008): Editor: X

Monitored behaviors

$\beta_d =$ "copy an executable file to a sensitive area"
$\beta_p =$ "copy to an area of your computer that shares files with others"
$\beta_m =$ "connect Internet in a suspicious manner to send out mail"
$\beta_l =$ "copy to multiple locations"
$\beta_r =$ "attempt to register itself in your Windows system startup"

Number of executions	Detected behaviors	Detection rate
500	{}	44 (8.8%)
	$\{\beta_m\}$	80 (16%)
	$\{\beta_d, \beta_l\}$	16 (3.2%)
	$\{\beta_p, \beta_l\}$	140 (28%)
	$\{\beta_m, \beta_l\}$	16 (3.2%)
	$\{\beta_m, \beta_r\}$	32 (6.4%)
	$\{\beta_d, \beta_p, \beta_l\}$	68 (13.6%)
	$\{\beta_d, \beta_m, \beta_l\}$	20 (4%)
	$\{\beta_p, \beta_l, \beta_r\}$	48 (9.6%)
	$\{\beta_d, \beta_p, \beta_l, \beta_r\}$	28 (5.6%)
	$\{\beta_d, \beta_m, \beta_l, \beta_r\}$	8 (1.6%)

each. No common patterns could be found to help understanding the detection support. In addition, contrary to P2P shared directories, no monitoring seems to be deployed on mail activity in order to detect its suspicious use for propagation (in particular for those labelled as Trojans).

5.3.4 Product B results [1]

Product B also relies on action monitoring but contrary to product A which searches for a global generic behavior (P2P Worms, Viruses, Trojans, etc.), product B looks for individual fine-grained suspicious behaviors as described in Table 4. For each detected behavior, the user is warned and asked for a decision: by default we have accepted all operations in order to continue the detection process (for this reason, the results have been gathered according to the different behavior combinations). At first glance, the results are quite promising with an excellent coverage. Only duplication seems to be problematic (28, 0% of detection for β_d whereas it is present in 100% of the variants). This can be explained by the fact that only sensitive areas are monitored, that is to say the system directories. A second explanation, which is also valid for propagation through P2P shared directories, is that standard C primitives, different from the Windows standard ones, can be used in order to bypass the engine. On the other hand, every attempt to propagate through mail has been detected without exception. With regards to residency, all attempts to register through a run registry key have also been detected but none of the other techniques.

This product offers the best coverage even if the ideal case would be the detection of the four behaviors at every execution (Mail variants: $\{\beta_d, \beta_m, \beta_l, \beta_r\}$ and P2P variants: $\{\beta_d, \beta_p, \beta_l, \beta_r\}$). Some additional tests, run in the next section, will be interesting to check that these good results do not result in an exacerbated false positive rate. In practice, no correlation is done between these behaviors which would help to identify generic threats in case of repeated erroneous decisions from the user.

5.4 Global evolution in behavioral detection

Through the tested products, we were partially satisfied to notice an evolution from our first evaluation two years ago [14]. According to these previous tests, we had come to the conclusion that either behavioral detection was unused by antivirus products or behavioral detection was severely hindered by its correlation with signature-based detection. This situation no longer seems to be in practice and the tests have shown a real deployment of behavioral detection even if some progress remains to be achieved with regards to the behavioral signatures and models.

Another global observation put forward by this test procedure is the diversity in the techniques of behavioral detection chosen from an editor to another. No single detection solution has really superseded the others. This observation is also relevant with regards to the behavioral models: according to the products, the behavioral models can be global ones with generic classes of malware or fine-grained with individual behavior descriptions (duplication, residency, mail propagation, P2P propagation). This can be explained by the fact that behavioral detection is still a recent and active research field producing new results every year.

Globally, finer-grained behavior models exhibit the best results, however like we said previously, these results must be crossed with the resulting false positive rates. In a second step of the procedure, we have confronted these same four products to different programs whose activity could raise some suspicions. A list of these programs as well as the obtained results are gathered in Table 5. A first observation is that individual behavior models suffer from greater rates of false positives ($\beta_{fp_1}, \beta_{fp_4}, \beta_{fp_6}$) as a drawback for their good detection rate. To cope with these false positives, the product B seems to use white-listing as a solution for known legitimate programs (β_{fp_6}). Notice that this white-list is established according to the executable names, which can obviously be easily bypassed. With regards to the other products using global behavioral models, the number of false positives is almost null and the raised alerts are no real false positives. First because KaZaA is well known to contain an incalculable number of bundled Spyware and Adware (β_{fp_7}). Then, because during the installation of antiviral products, the monitoring techniques which are deployed, are identical to the hooking and stealth techniques used in malware ($\beta_{fp_2}, \beta_{fp_3}, \beta_{fp_4}$). We think that, the main point with global approaches, in addition to their low detection rates, would be some naive policies which proves too generic in application: radically blocking SMTP port is a trivial example (β_{fp_5}).

Table 5 Assessment of the false positive rates. See Tables 1, 2, 3 and 4 for the product references

Program	Context	DrWeb	NOD32	Prod. A	Prod. B
Explorer	Run				β_{fp_1}
Patch DNS (KB945553)	Install				
AV product	Install			β_{fp_2},β_{fp_3}	β_{fp_4}
Office XP	Install				
	Run				
Telnet	Run		β_{fp_5}		
mIRC	Install				
	Run				β_{fp_6}
Skype	Install				
	Run				
FtpExpert3	Install				
	Run				
KaZaA	Install	β_{fp_7}	β_{fp_7}	β_{fp_7}	
	Run				

False positives

β_{fp_1} = "attempting execution of instructions from an unauthorized area"(launching executables from the explorer in miniature view)

β_{fp_2} = "suspicious driver installation to get overall access to the system"

β_{fp_3} = "invader attempting to insert in winlogon"

β_{fp_4} = "modify the way your computer communicate with the Internet"

β_{fp_5} = "some useful ports are completely blocked (ex. SMTP 25)"

β_{fp_6} = "potentially unwanted application that may exhibit malware characteristics, mirc.exe: known as not a virus"

β_{fp_7} = "alerts concerning various Spyware and Adware"

6 Use case in software protection

It is not really surprising that, the techniques for software protection and the techniques used in malware to mutate and thwart analysis, are strongly linked. The purpose is basically the same. Malware creators often use these techniques to slow down the analysis process which is led by experts in order to extract a signature or information to identify the attack. The only difference lies in the time available to analyze the code between a hacker and an expert overwhelmed by thousands of variants. We think that functional polymorphic engines provide interesting features with respect to software protection:

Static analysis: The control flow graph of the effective code is only written during execution. The control flow directly depends on the randomly chosen annotated derivation tree. This means that even if a hacker use an emulator to collect the generated code, he will only collect a single version among several equivalent variants. Besides, this building respects an important principle in anti-tampering protection that is the dependence between the control flow and the data flow [26]. Here the code structure and control directly depends on random data generated during derivation. Trying to address the analysis of the engine itself, the hacker will be confronted to an important amount of alternative execution paths in the derivation and translation modules. The number of branching is actually proportional to the entropy calculated in Sect. 3.2.

Dynamic analysis: Once again, the code is only written during execution and it weights heavily on dynamic analysis in particular with regards to breakpoints. Independently from the execution level of the debugger (ring 0 or ring 3), the hacker does not know exactly where the code will be built in memory until the allocation. Moreover, the code will be different from an execution to an other, meaning that the predicted location of the breakpoint is likely to be at the wrong address, possibly unaligned with the assembly code.

Limitations: The main drawback from these engines is that they introduce an original overload explained by the code building. Consequently, functional polymorphic generation should be restricted to limited critical portions of code, but sufficiently important to offer enough possible variations. In addition, just like any other anti-tampering technique, these engines exhibit some weaknesses. The security of the scheme relies on the difficulty to establish a correspondence between the original point of the derivation (the start symbol) and the purpose of the generated code. This correspondence is hard to tell because of the numerous intermediate functions implicated in derivation, but it could be found more easily using forced branching instead of random branching during derivation. But using a combination of different anti-tampering techniques, they can consolidate each other. In particular, dynamic integrity checking [27] and anti-debug techniques could thwart forced branching. The implications of functional polymorphic engines in software protection have been briefly described here to argue their potential uses but they should be explored in greater details.

7 Conclusion

7.1 Contribution and ethical considerations

In this paper, we have introduced the new concept of automated functional mutations from both the theoretical perspective and the operational perspective. The functional polymorphism engines are simply the automation of what most malware writers actually do: to take a known strain and slightly modify their functionalities to avoid detection. We did not intend to make their task easier. The fact is that we were more interested in the possible applications for security researchers and experts. In particular, we have put forward two possible use cases: for behavioral detectors assessment by simulation of unknown malware using known techniques and for software protection by dynamic generation of variable code. In practice, an important amount of work remains before offensive malware can be obtained from our engine. We have only a limited set of the most common behaviors at our disposal (no complex payload for example), and these behaviors are all based on existing malicious techniques meaning that they remain detectable. In addition, the engine itself could easily be detected by signature just like decryption routines in polymorphism.

7.2 Perspectives and solutions

The perspective is now to make the engine richer with additional behaviors but also to increase the number of possible derivations with new semantic attributes. These enhancements should result in a greater completeness of our test procedure for behavioral detectors. On the opposite, detection should also benefit from this work. Basically, functional polymorphism engines and behavioral detectors have an inverse functioning: a mutation engine implements an abstract model into binary code for execution whereas the detector translates execution information into an abstract description for comparison to a model. Therefore, a translation mechanism could prove useful to generate new behavioral signatures with a better coverage than the one used in the tested products. Current works are in progress in order to develop such an analyzer based on behavioral grammars.

Acknowledgments This work has been partially supported by the European Commissions through project FP7-ICT-216026-WOMBAT funded by the 7th framework program. The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the European Commission.

References

1. Jacob, G., Debar, H., Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy (WTCV'07 special issue). J. Comput. Virol. **4**(3) (2008)
2. Fortinet Observatory. http://www.fortinet.com/FortiGuardCenter/
3. Virus Construction Tools From viruslist.com. http://www.viruslist.com/en/virusesdescribed?chapter=153318618
4. Vx heaven: virus creation tools repository. http://vx.netlux.org/vx.php?id=tidx
5. Wagner, D., Soto, P.: Mimicry attacks on host based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security (2002)
6. Gao, M.K.R.D., Song, D.: On gray-box program tracking for anomaly detection. In: Proceedings of the 13th USENIX Security Symposium, pp. 103–118 (2004)
7. Driller, T.M.: Metamorphism in practice, 29A E-zine, vol. 6 (2002)
8. Filiol, E.: Techniques Virales Avancées. Springer, IRIS Collection, ISBN: 2-287-33887-8 (2007)
9. Ször, P.: The Art of Computer Virus Research and Defense. Addison-Wesley, Reading, ISBN:0-321-30454-3 (2005)
10. Beaucamps, P.: Advanced polymorphic techniques. Int. J. Comput. Sci. **2**(3), 194–205 (2007)
11. Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. In: Proceedings of the International Conference on Computational Intelligence (ICCI), Published in Int. J. Comput. Sci. **2**(1) 70–75 (2007)
12. Spinellis, D.: Reliable identification of boundedlength viruses is np-complete. IEEE Trans. Inf. Theory **49**(1), 280–284 (2003)
13. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantic-based approach to malware detection. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (2007)
14. Filiol, E., Jacob, G., Liard, M.L.: Evaluation methodology and theoretical model for antiviral behavioural detection strategies (WTCV'06 special issue). J. Comput. Virol. **3** (1) 23–37 (2007)
15. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages and Computation, 2nd edn. Addison Wesley, New York, ISBN:0-201-44124-1 (1995)
16. Knuth, D.E.: Semantics of context-free grammars. Theory Comput. Syst. **2**(2), 127–145 (1968)

17. Noll, T.: Compiler construction, lectures 15 to 18: Semantic analysis. RWTH Aachen University (2006). http://www-i2.informatik.rwth-aachen.de/Teaching/Course/CB/2006/Slides/
18. Shannon, C.E.: A mathematical theory of communications. Bell Syst. Tech. J. **27**, 379–423, 623–656 (1948)
19. Filiol, E.: Malware pattern scanning schemes secure against black-box analysis (EICAR 2006 special issue). J. Comput. Virol. **2**(1), pp. 35–50, (2006)
20. Papadimitirou, C.H.: Computational Complexity. Addison Wesley, Reading. ISBN:0-201-53082-1 (1995)
21. Jacob, G., Filiol, E., Debar, H.: Malwares as interactive machines: a new framework for behavior modelling (WTCV'07 special issue). J. Comput. Virol. **4**(3) (2008)
22. Charlier, B.L., Mounji, A., Swimmer, M.: Dynamic detection and classification of computer viruses using general behaviour patterns. In: Proceedings of the 5th Virus Bulletin Conference (1995)
23. Ford, R., Wagner, M., Michalske, J.: Gatekeeper ii: new approaches to generic virus prevention. In: Proceedings of the 14th Virus Bulletin Conference (2004)
24. Christodorescu, M., Jha, S.: Testing malware detectors. In: Proceedings of ACM SIGSOFT: Internatinal Symposium on Software Testing and Analysis (ISSTA 04), pp. 34–44 (2004)
25. Qemu: open source processor emulator. http://fabrice.bellard.free.fr/qemu/
26. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: obstructing static analysis of programs. Tech. Rep. CS-2000–2012 (2000)
27. Horne, B., Matheson, L.R., Sheehan, C., Tarjan, R.E.: Dynamic self-checking techniques for improved tamper resistance. In: Proceeding of the Digital Rights Management Workshop pp. 141–159 (2001)