# Viral Logical Concept Analysis for Malware Conceptual Hierarchy Generation

Nguyen Thien Binh, Tran Cong Doi, Quan Thanh Tho, and Nguyen Minh Hai

***Abstract*—Automatic classification of virus samples into a concept hierarchy has been attracting much attention from malware research community. This would help anti-virus experts to have an obvious and systematic view on the landscape of virus samples, whose numbers have been rapidly increasing recently. However, it is not a trivial work, since malwares usually come in binary forms whose actions are complicated and obfuscated. Therefore, the typical data mining approaches based on feature extraction are not easily applied.**

**In this paper, we introduce an approach using Formal Concept Analysis (FCA) to generate a malware hierarchy. Since virus behaviours are often described effectively by temporal logic, we extend formal paradigm of FCA by using Logical Concept Analysis (LCA), where concepts are generalized by logic formulas. We also enhance the basic LCA to Viral Logical Concept Analysis (V-LCA), where abstraction techniques are used to abstract formal concepts representing virus samples. Our approach has been applied in a real dataset of virus and promising experiment results have been acquired.**

***Index Terms*—Computer virus, malicious software, malware detection, formal concept analysis, logical concept analysis, viral logical concept analysis, conceptual clustering.**

## I. Introduction

### *A. Computer Virus*

*Computer virus* (from now we call *virus*), or *malware*, is a segment of computer programs which executes actions to harm to a computer system potentially. When infecting a file, virus also copies a unique syntactic pattern, known as *signature*, to the file. When a virus discovers this pattern from a file, it recognizes that this file is infected and does not replicate itself. Based on this characteristic, most of industry anti-virus programs detect virus by scanning whether signature appears or not. However, this method has difficulty in dealing with advanced viruses such as *polymorphic* and *metamorphic* virus [1], [2] because these viruses virtually create different signature after each infection.

To solve this problem, recent studies have suggested a method of virus detection based on determining hazardous behavior instead of matching pattern [3], [5]. For example, let us consider some code fragment as shown in Table I. There are 6 code fragments, referred as *A*, *B*, *C*, *D*, *E* and *F*. Fragment A is in fact the well-known *Avron* virus. Its harmful behaviors include pushing *zero* to the top of stack (by assigning zero for register *ebx* and pushing it to the top of stack), then the virus body will be executed to invoke *GetModuleFileNameA* in order to get the name and path of the victim file. Then the virus will proceed to replace the original code in the victim by the malicious code of the virus itself. Thus, each time the victim file is executed on a computer, it will infect virus on the whole system. This process will be ongoing and the number of infected computers will increase rapidly. Hence, even though Avron has several variants, which have different signatures, the *behavior* of finding the name and path of the victim by means of *GetModuleFileNameA* is still always remained.

Fragment B shows a variant of this virus, which uses *ecx* instead of *eax*. Fragment *C* shows another variant of this virus, in which XOR is executed to assign zero for *ebx*. Fragment *F* shows a sophisticated variant, in which the assignment and call stack instruction are replaced with instruction that accesses stack via pointer and instruction that jumps to entry address of function. Particularly, in variants *C* and *F*, the viruses use *obfuscation techniques* of *junk code,* which insert some instructions that do not make sense, such as *inc a* or *dec b*, where *a* and *b* are two dumping variables[1].

Meanwhile, there are no virus found in Fragment *D* and Fragment *E*, but the instructions involved on those pieces of code are also quite similar to the discussed virus samples.

### *B. Temporal Logic to Represent viral Behaviours*

Since a virus sample can be morphed into multiple variants as previously discussed, research community tends not to represent virus by syntactic patterns. Instead, logic is then proposed to capture viral behaviors. This means each virus sample is represented by a logical formula. *Temporal Logic* (TL) [6] is used commonly due to its capability of describing correctly the execution sequence of virus behaviors. For example, the code fragments in Table 1 can be represented by a TL formula of *F(mov(ebx,0)∧Xpush(ebx)∧Xcall(GetModuleFileNameA)*, in which the operator *F* is understood as *Eventually*, and the operator *X* as *Next*. This logic formula can be interpreted as: "*in a binary code, if eventually there is an instruction assigning 0 for register ebx, and then value of ebx is pushed to stack and subsequently GetModuleFileNameA function is called, this binary code is infected by Avron virus*". Likewise, temporal logic formulas for other code fragments are also presented, as depicted in Table I.

Manuscript received April 27, 2017; revised August 13, 2017. This research is funded by Vietnam National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.01-2015.16.

B. T. Nguyen, T. T. Quan, and H. M. Nguyen are with Ho Chi Minh City University of Technology, Vietnam (e-mail: 551105019@stu.hcmut.edu.vn, qttho@hcmut.edu.vn, 551307910@hcmut.edu.vn).

D. C. Tran is with Dong Nai University, Vietnam (e-mail: congdoivc@gmail.com).

[1] Detailed discussion on obfuscation technique is beyond the scope of this paper.

### *C. Generate Virus Concept Hierarchy Using FCA-based Approach*

As the number of viruses is increasing rapidly, the demand of automatic classification of malware becomes highly desired. For example, an expert can classify the virus sample in Table I as a *concept hierarchy* as described in Fig. 1. One can observe that the code segments are organized into two groups of *Avon family* malware and *benign* program. In the rest of the paper, we will discuss on how to construct such a hierarchy automatically from a set of code fragments.

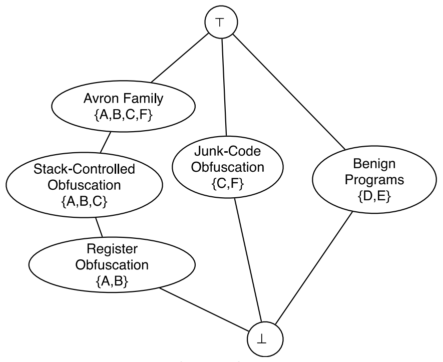

⊤
Avron Family {A,B,C,F}
Stack-Controlled Obfuscation {A,B,C}
Junk-Code Obfuscation {C,F}
Benign Programs {D,E}
Register Obfuscation {A,B}
⊥

Fig. 1. Concept hierarchy of viral fragments described in Table I.

To build such a hierarchy from a large dataset of viruses. Formal Concept Analysis (FCA) [7] is a suitable technique, because this method supports to construct a concept lattice representing the hierarchical relationships among sets of objects in dataset, each of which is represented by a set of attributes. Moreover, FCA has strong support for *conceptual clustering*, in which objects are grouped to form real-life concepts.

However, since viruses represented by logical formulas, as discussed, using FCA suffers from the difficulty that the set of attributes used in FCA does not logically reflect the behaviours captured by the logical formula. An extension of FCA, known as *Logical Concept Analysis* (LCA), allows to generalize each object in the hierarchy by a logical formula. Thus, this approach seems suitable to represent each virus sample by a formula.

In this paper, we propose an approach based on LCA in order to generate concept hierarchy automatically from a virus dataset. In particular, we extend LCA to *Viral Logical Concept Analysis* (V-LCA), where abstraction techniques, suggested by other works in malware research, are adopted generalize new concepts from existing one, in order to form a *viral lattice*. Finally, we perform conceptual clustering on this logic viral lattice to obtain the final malware conceptual hierarchy.

The rest of this paper is organized as follows. Section II reviews some related works. In Section III and IV, we present V-LCA and the conceptual clustering technique. Section V discusses our experiments. Finally, Section VI concludes the paper.

TABLE I: SOME CODE SEGMENTS ILLUSTRATING THE VIRAL BEHAVIORS

| ID | Sample Pattern | Logic Formulas | Meaning |
|---|---|---|---|
| *A* | *mov ebx,0*<br>*push ebx*<br>*call GetModuleFileNameA* | ***F****(mov(ebx,0)∧****X****push(ebx)∧* ***X****call(GetModuleFileNameA)* | Avron virus |
| *B* | *mov ecx,0*<br>*push ecx*<br>*call GetModuleFileNameA* | ***F****(mov(ecx,0∧****X****push(ecx)∧* ***X****call(GetModuleFileNameA)* | Avron variant |
| *C* | *xor ebx,ebx*<br>*push ebx*<br>*inc a*<br>*call GetModuleFileNameA* | ***F****(xor(ebx,ebx)∧****X****push(ebx)∧****X****inc(a)∧* ***X****call(GetModuleFileNameA)* | Avron variant with junk code |
| *D* | *mov ebx,0*<br>*push ebx*<br>*push 1*<br>*call GetModuleFileNameA* | ***F****(mov(ebx,0)∧****X****push(ebx)∧****X****push(1)∧* ***X****call(GetModuleFileNameA))* | Not a virus |
| *E* | *call GetModuleFileNameA*<br>*push ebx*<br>*mov ebx,0* | ***F****(mov(ebx,0)∧****X****call(GetModuleFileNameA)* *∧****X****push(ebx))* | Not a virus |
| *F* | *sub esp, 4*<br>*mov [esp], 0*<br>*dec b*<br>*jmp GetModuleFileNameA* | ***F****(sub(esp,4)∧****X****mov([esp],a)∧****X****dec(b)∧* ***X****jmp(GetModuleFileNameA))* | Avron variant with junk code and stack obfuscation |

## II. RELATED WORKS

### *A. Formal Concept Analysis*

Formal Concept Analysis [7] is a data analysis technique aiming at recognizing formal concepts of a formal context and constructing a concept lattice accordingly.

In order to construct concept lattice more efficient and meaningful, many enhancements have been proposed. Zhang (2013) [8] combined FCA, Chu space and Domain Theory to analyze the dependency among the attributes. [9] proposed using closure operator to analyze the dependency among the attributes. Concept location method was proposed by [10]. In work of [11], an algebraic structure was proposed to build the concept hierarchy and ontology merging from concept lattice.

When using the FCA technique for conceptual clustering, the challenge is the contextual implication of the attributes to represent the concept [12]. [13] used the set of attributes to make preference models, hence introducing the *ceteris paribus preferences*. Using linguistic hedges have been proposed by [14].

However, while the domain concepts are represented by the complex logic behaviors such as the malicious actions of virus, the concept representation requires techniques to analyze and handle the relationship between the complex attributes, which remains unresolved by existing works.

### *B. A Logical Generalization of Formal Concept Analysis*

Since the number of formal concepts generated by FCA is huge, many approaches were proposed to generalize a set of similar concept forms on the concept lattice to create *a generalized representation* for similar concepts. Therefore, the number of concepts on the lattice is reduced and the concepts are also more comprehensible [15], [16].

One of the most common approaches on this direction is *logical generalization* [17], which represents a set of similar concepts by a logic formula. The advantage of this approach is that logic formula can be further processed in an automatic manner for more discovered knowledge. In [18], concepts of FCA are extracted and generalized into logic description. At a higher level, logical axioms are extracted by finite interpretations on concept lattice [18].

However, to describe virus behavior, *temporal logic* is a better choice because it can represent and verify a sequence of execution actions as illustrated in motivating examples. Moreover, the generalization of temporal logic formulas in this case requires an abstract interpretation mechanism to connect those temporal logic formulas following the operational mechanism of machine code. Dealing with this challenge is also a contribution of our papers.

## III. Viral Logic Concept Analysis

### A. Formal Concept Analysis

**Definition 1 (Formal context).** A *formal context* is a triple $K=(G, M, I)$ where $G$ is a set of objects, $M$ is a set of attributes, and $I$ is a set of binary relations between $G$ and $M$, $I \subseteq G \times M$. An object $g \in G$ has an attribute $m \in M$ is denoted as $(g,m) \in I$ or $g\ I\ m$.

Table II represents formal context made from code fragments of Table I. Each code fragment now is considered as an object, and machine code instructions used in those fragments are the attributes on this context.

TABLE II: Formal Context Created from the Segment of Programs in Fig. 1

| | mov | ebx | push | call | moduleA | ecx | xor | inc | a | sub | esp | [] | dec | b | jmp | 1 | 0 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | x | x | x | x | x | | | | | | | | | | | | x | |
| B | x | | x | x | x | x | | | | | | | | | | | x | |
| C | | x | x | x | x | | x | x | x | | | | | | | | | |
| D | x | x | x | x | x | | | | | | | | | | | x | x | |
| E | x | x | x | x | x | | | | | | | | | | | | x | |
| F | x | | | | x | | | | | x | x | x | x | x | x | | | x |

**Definition 2 (Derivation operator).** In a context *(G,M,I)*, *derivation operator* is an operation that produces for a set of objects $A \subseteq G$ the set of attributes which are shared by all these objects and for a set of attributes $B \subseteq M$ the set of objects which share all these attributes, and is denoted as follows:

$$A \subseteq G,\ A' = \{m \in M \mid \forall g \in A, \exists(g,m)\}$$

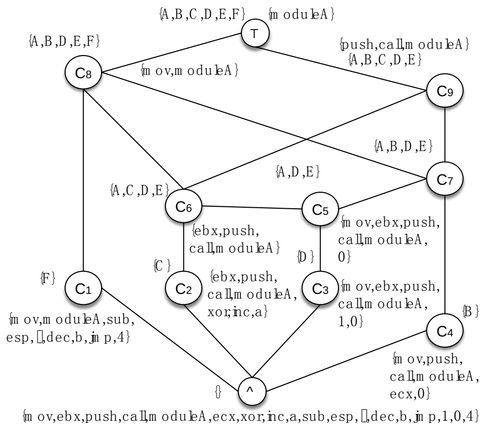
$$B \subseteq M,\ B' = \{g \in G \mid \forall m \in B, \exists(g,m)\}$$

**Definition 3 (Formal concept).** In a context $(G, M, I)$, $(A, B)$ is a *formal concept* if and only if $A \subseteq G$, $B \subseteq M$, $A=B'$, and $B=A'$. $A$ is called the extent of the formal concept $(A, B)$ and $B$ is called the intent of the formal concept $(A,B)$.

**Definition 4 (Concept relation).** In a context $(G,M,I)$, $(A_1B_1) \leq (A_2, B_2)$ if and only if $A_1 \subseteq A_2$ or $B_2 \subseteq B_1$. $(A_1B_1)$ is called the subconcept of the formal concept $(A_2, B_2)$ or $(A_2, B_2)$ is called the superconcept of the formal concept $(A_1B_1)$.

**Definition 5 (Concept lattice).** In a context $(G,M,I)$, the set of all formal concepts ordered by concept relations is called the *concept lattice*.

Fig. 2 describes concept lattice generated from the formal context depicted in Table II.



{A,B,C,D,E,F}
{moduleA}
T
{A,B,D,E,F}
C8
{mov,moduleA}
{push,call,moduleA}
{A,B,C,D,E}
C9
{A,B,D,E}
C7
{A,D,E}
{A,C,D,E}
C6
C5
{ebx,push, call,moduleA}
{mov,ebx,push, call,moduleA, 0}
{D}
{C}
{F}
C1
C2
{ebx,push, call,moduleA, xor,inc,a}
C3
{mov,ebx,push, call,moduleA, 1,0}
{B}
C4
{mov,moduleA,sub, esp,[],dec,b,jmp,4}
{mov,push, call,moduleA, ecx,0}
{}
^
{mov,ebx,push,call,moduleA,ecx,xor,inc,a,sub,esp,[],dec,b,jmp,1,0,4}

Fig. 2. The concept lattice of virus is generated by FCA method.

As shown in Fig. 2, three sample programs *A, D* and *E* will be grouped into a concept because they have the same attributes. It is not reasonable because only *A* is a virus in three programs. Moreover, the intent representation of each concept is not precise enough to determine a virus set. For example, in the $C_8$ concept, its intent of {*mov,ModuleA*} is too general since a normal program can also invoke those instructions as illustrated in the segment programs of *D* and *E*.

### B. Logical Concept Analysis

Based on the theory of FCA, [17] proposed the approach of Logical Concept Analysis (LCA) as follows.

**Definition 6 (Logic context).** *A* logic context is a triple $(\mathcal{O},\mathcal{L},i)$ where:

- $\mathcal{O}$ is a finite set of objects,
- $\langle \mathcal{L}; \models \rangle$ is a (possibly infinite) lattice of formulas, whose supremum is $\dot{\vee}$ and whose infimum is $\dot{\wedge}$; C denotes a logic whose deduction relation is $\dot{\models}$. and whose is junctive and conjunctive operations are respectively $\dot{\vee}$ and $\dot{\wedge}$,
- $i$ is a mapping from $\mathcal{O}$ to $\mathcal{L}$ that associates to each object a formula that describes the object.

TABLE III: Logic Formal Context of Code Fragments on Table I

| **Object** | **i(O)** |
|---|---|
| *A* | ***F****(mov(ebx,0)∧****X****push(ebx)∧****X****call(GetModuleFileNameA)* |
| *B* | ***F****(mov(ecx,0∧****X****push(ecx)∧****X****call(GetModuleFileNameA)* |
| *C* | ***F****(xor(ebx,ebx)∧****X****push(ebx)∧****X****inc(a)∧****X****call(GetModuleFileNameA)* |
| *D* | ***F****(mov(ebx,0)∧****X****push(ebx)∧****X****push(1)∧****X****call(GetModuleFileNameA))* |
| *E* | ***F****(mov(ebx,0)∧****X****call(GetModuleFileNameA)∧****X****push(ebx))* |
| *F* | ***F****(sub(esp,4)∧****X****mov([esp],0)∧****X****dec(b)∧****X****jmp(GetModuleFileNameA))* |

**Definition 7 (Logic concept).** In a context $(\mathcal{O},\mathcal{L},i)$, a *logic concept* is a pair $c = (O,f)$ where $O \subseteq \mathcal{O}$. and $f \in \mathcal{L}$, such that $\sigma(O)=f$ and $\tau(f) = O$. The set of objects $O$ is the concept *extent* (written *ext(c)*), whereas formula $f$ is its *intent* (written *int(c)*).

The main difference of LCA to standard FCA is that the intent is now a formula of the logic *C*. The set of all concepts that can be built in a context $(\mathcal{O},\mathcal{L},i)$ is denoted by $\mathcal{C}(\mathcal{O},\mathcal{L},i)$ and is partially ordered by $\leq^c$ defined as follows.

**Definition 8 (Order $\leq^c$).** *Let* $(O_1,f_1)$ and $(O_2,f_2)$ be in $\mathcal{C}(\mathcal{O},\mathcal{L},\text{i})$,

$$(O_1,f_1) \leq^c (O_2,f_2) \Leftrightarrow O_1 \subseteq O_2$$

This order is compatible with order on intents.

**Proposition 1.** $(O_1,f_1) \leq^c (O_2,f_2) \Leftrightarrow (f_1 \vDash f_2)$

From the order stated in Proposition 1, one will get the corresponding logic. Fig. 3 describes logic lattice generated from logic context in Table III.

However, in order to do so, one needs a method to build *generalized concept* $C'=(O',f')$ from a set of concepts $\{C_1=(O_1,f_1),..,C_n=(O_n,f_n)\}$ such that $C_i \subseteq C'$ $(\forall i)$. For example, in the lattice of Fig. 3, two concepts $C_1$ and $C_2$ are generalized into concept $C_5$. This is implemented by a special operation of $\uplus$, *known as widening operator*, such that $f' = f_1 \uplus ... \uplus f_n$ and $f' \vDash f_i$. $\forall i$.

The operation $\uplus$ will be implemented depending on the domain on which LCA is applied. In the next section, we will discuss how to implement such operations for temporal logic representing virus behaviors.

## C. Viral Logic Concept Analysis

In this section, we present Viral Logical Concept Analysis (V-LCA), an extension of LCA to produce a malware lattice from dataset of virus programs. The main idea of V-LCA is to use *temporal logic* to capture intents of concepts. Since temporal logic allows representing the *execution order* of execution of the operations, it is very suitable to represent virus behaviors. Then, we apply *viral abstractions* to implement the widening operator previously discussed. Viral abstraction represents a set of temporal formula representing variants of the same virus samples by a new generalized formula.

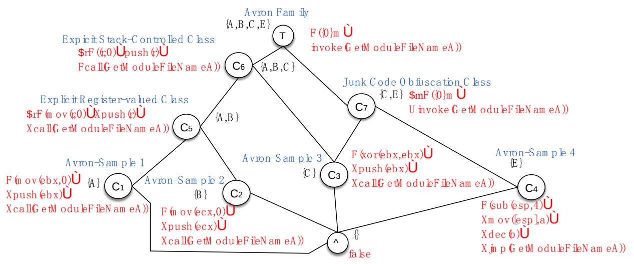

Avron Family
{A,B,C,E}
T
F({0}m Ù
invoke(GetModuleFileNameA))
Explicit Stack-Controlled Class
$rF((r,0)Ùpush(r)Ù
Fcall(GetModuleFileNameA))
C6
{A,B,C}
Junk Code Obfuscation Class
{C,E}
$mF({0}m Ù
U invoke(GetModuleFileNameA))
C7
Explicit Register-valued Class
$rF(mov(r,0)ÙXpush(r)Ù
Xcall(GetModuleFileNameA))
C5
{A,B}
Avron-Sample 1
F(mov(ebx,0)Ù
Xpush(ebx)Ù
Xcall(GetModuleFileNameA))
{A}
C1
Avron-Sample 2
{B}
C2
F(mov(ecx,0)Ù
Xpush(ecx)Ù
Xcall(GetModuleFileNameA))
Avron-Sample 3
{C}
C3
F(xor(ebx,ebx)Ù
Xpush(ebx)Ù
Xcall(GetModuleFileNameA))
Avron-Sample 4
{E}
C4
F(sub(esp,4)Ù
Xmov([esp],a)Ù
Xdec(b)Ù
Xjmp(GetModuleFileNameA))
^
{}
false

Fig. 3. The concept lattice generated by V-LCA.

**Definition 9 (Viral Logic Formal Context).** *A* viral logic formal context is a triple *$(\mathcal{O},\mathcal{L},i)$* where*:*

- $\mathcal{O}$ is a finite set of objects, each of which represents a virus instance or a set of virus instances,
- $\langle \mathcal{L}; \vDash \rangle$ is lattice of temporal formulas, whose supremum is $\dot{\vee}$ and whose infimum is $\dot{\wedge}$,
- i *is a* mapping from $\mathcal{O}$ *to* $\mathcal{L}$ that associates to each object a formula that describes virus(es) behaviors.

In order to generate a *viral lattice*, one needed widening operator to generate super concepts from subconcepts. As discussed, we use viral abstractions to implement this widening operator, as shown in Table IV.

**Example 1.** Given two formulas of *EF((mov ebx,0)∧AF(push ebx)∧AXcall(GetModuleFileNameA))* and *EF((mov ecx,0)∧AF(push ecx)∧ AXcall(GetModuleFileNameA))*, they are almost the same but only using two different registers of *ebx* and *ecx*. The viral abstraction *CTPL* will generate new generalized formula ∃ *rEF(mov(r,0)∧AFpush(r)∧AXcall(GetModuleFileNameA))* where *r* stands for an arbitrary register.

**Example 2.** Given the formulas of *EF((mov ebx,0)∧AF(push ebx)∧AX call(GetModuleFileNameA))*, *EF((mov ecx,0)∧AF(push ecx)∧AXcall(GetModuleFileNameA))*, *EF(xor(ebx,ebx)∧AFpush(ebx)∧AFinc(a)∧AXcall(GetModuleFileNameA)* and *∃rEF(mov(r,0)∧AFpush(r)∧AXcall(GetModuleFileNameA))*, all formulas are in fact doing the same thing of pushing 0 the top stack. All of them are viral abstracted by SCTPL\X to generate new formula *∃mEF(call(GetModuleFileNameA)∧{0}mΓ$^*$ )* where the notation of *{0}mΓ$^*$* implies the value of top stack *Γ$^*$* is *m* and value of *m* is *0*.

TABLE IV: VIRAL ABSTRACTION

| No | Viral Abstraction | Description | Example |
|---|---|---|---|
| 1. | **CTPL [3]** | When two temporal logic formulas are only different in terms of register usage, they can be abstracted by a new formula where the register is represented by a variable | Two formulas *EF((mov ebx,0)∧AF(push ebx)∧AX call(GetModuleFileNameA))* and *EF((mov ecx,0)∧AF(push ecx)∧AXcall(GetModuleFileNameA))* will be abstracted as ∃*r EF(mov(r,0)∧AFpush(r)∧AXcall(GetModuleFileNameA))* |
| 2. | **SCTPL [4]** | Abstraction of the action of pushing a value into top of stack | Two formulas *EF((mov ebx,0)∧AF(push ebx)∧AX call(GetModuleFileNameA))*, *EF((mov ecx,0)∧AF(push ecx)∧AX call(GetModuleFileNameA))* and ∃*rEF(mov(r,0)∧ AFpush(r)∧AXcall(GetModuleFileNameA))* will be abstracted as *∃rEF(mov(r,0)∧ AX(call(GetModuleFileNameA)∧rΓ$^*$))*, showing that the value of top stack is currently *r* |
| 3. | **SCTPL\X [5]** | Actions that change contents of stack will be abstracted by an expression directly modifying stack contents | The formulas *EF((mov ebx,0)∧AF(push ebx)∧AX call(GetModuleFileNameA))*, *EF((mov ecx,0)∧AF(push ecx)∧AX call(GetModuleFileNameA))*, *∃rEF(mov(r,0)∧ AFpush(r)∧AXcall(GetModuleFileNameA))*, and *∃rEF(mov(r,0)∧AX(call(GetModuleFileNameA)∧rΓ$^*$ ))* will be abstracted as *∃mEF(call(GetModuleFileNameA)∧{0} mΓ$^*$ )*, showing that the stack top is storing *m*, whose value is *0* |

## IV. LLCC: Logic-Based Lattice Conceptual Clustering Algorithm

In previous section, we present using V-LCA to generate the viral lattice, where the formal concepts are organized as a hierarchical structure. However, since the number of the logic concept is quite large when applied in practice, we propose using conceptual clustering to cluster similar formal concepts. As a result, we have a concept hierarchy of malware as depicted in Fig. 1. Aiming at generating a concept hierarchy, most of conceptual clustering methods are based on the method *hierarchical agglomerative clustering* (HAC), such a COBWEB [19]. The order approach is based on FCA result, such as FOGA [20]. However, those algorithms always suffer from the performance problem due to the high complexity of $N^2$. In our case, the formal concepts generated V-LCA are represented by logic formulas and ordered by a partial order. We propose an enhanced conceptual clustering algorithm known as Logic-based Lattice Conceptual Clustering Algorithm (LLCC).

Details of the algorithm can be described as follows.

```
For each object o in the formal context
        For each concept c on current concept lattice
                Add o as a concept of the current lattice
                If o ⋈ c = o’ ≠ ε (empty) then
                If o’has the same featured formula then add o to c
Else introducing new concept c’
```

In order to construct the final featured concept lattice from a given featured formal context, obviously one needs to consider “grouping” objects into concepts. To do so, we introduce the object-joining operator $\bowtie$ on two featured concepts $C_1 = (A_1,I_1,\alpha_1)$ and $C_2 = (A_2,I_2,\alpha_2)$ is defined as follows: $C_1 \bowtie C_2 = \{C^*, I^*, \alpha^* = \alpha_1 \uplus \alpha_2\}$ where $\uplus$ is *widening operator* two formulas $\alpha_1$ and $\alpha_2$, $C^* = \{x \mid x \in (C_1 \cup C_2)$ *and* $\alpha^* \vDash \varphi(x)\}$ and $I^* = \{i \mid \exists g \in C^*: g \circ i\}$

In Definition 9, $\uplus$ is *widening operator* two formulas $\alpha_1$ and $\alpha_2$. The simplest widening operator is the operator $\vee$, which is the disjunction operator in the case of propositional logic, and is more commonly known as *least common subsumer operator* $\cup$ when processing concepts on the concept lattice. However, for temporal logic representing virus behavior, *widening operator* include viral abstraction as previously discussed. Hence, a concept of the form $A \cup B$ can be further widened as an abstract concept $C$ if existing a viral abstraction, which can abstract both $A$ and $B$ to $C$.

**Example 3.** When one performs $C_5 = C_1 \bowtie C_2$ as illustrated in Fig. 3, firstly the *least common subsumer* $F(mov(ebx,0) \wedge Xpush(ebx) \wedge Xcall(GetModuleFileNameA)) \cup F(mov(ecx,0) \wedge Xpush(ecx) \wedge Xcall(GetModuleFileNameA))$ is performed. Then, since existing the formula $\exists r\, F(mov(r,0) \wedge Xpush(r) \wedge Xcall(GetModuleFileNameA))$ can be abstracted from both $F(mov(ebx,0) \wedge Xpush(ebx) \wedge Xcall(GetModuleFileNameA))$ and $F(mov(ecx,0) \wedge Xpush(ecx) \wedge Xcall(GetModuleFileNameA))$, we obtain the final result of $\varphi(C_5) = f(C_1) \uplus f(C_2) = \exists r F(mov(r,0) \wedge Xpush(r) \wedge Xcall(GetModuleFileNameA))$.

TABLE V: PERFORMANCE RESULTS BASED ON AUP

| | 150 | 200 | 250 | 500 | 1000 | 1500 | 2000 | 2500 | 3000 |
|---|---|---|---|---|---|---|---|---|---|
| HAC-based | 0.78 | 0.74066667 | 0.69871111 | 0.662 | 0.63053333 | 0.60693333 | 0.56235556 | 0.53613333 | 0.49942222 |
| FCA-based | 0.83 | 0.79066667 | 0.75133333 | 0.7172444 | 0.67791111 | 0.6412 | 0.60711111 | 0.57826667 | 0.5389333 |
| V-LCA | 0.976 | 0.94453333 | 0.91568889 | 0.8816 | 0.84226667 | 0.80293333 | 0.77671111 | 0.74524444 | 0.74 |

## V. EXPERIMENTS

### A. Dataset and Malware Hierarchy

We performed experiments by generating malware concept hierarchy from a dataset of 3000 virus samples downloaded from VXHeaven[2]. From this dataset, we constructed the concept hierarchy using baseline method of traditional FCA-based conceptual clustering and our proposed V-LCA-based conceptual clustering. The constructed Malware Hierarchy is illustrated as Fig. 4.

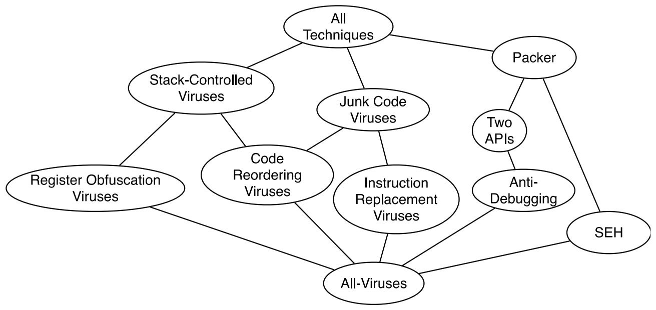


Fig. 4. The Malware Hierarchy.

### B. Performance Evaluation

We used *Mean Average Precision* (MAP) [21] that is defined as the mean of the *precision* value at each point (or node) in a hierarchical structure where a relevant item appears, divided by the total number of retrieved item. Typically, MAP implies the goodness of a concept hierarchical structure. If the structure is reasonable enough, when a node (i.e. a cluster) is retrieved against a query, all of objects belonging to this cluster should be relevant to the query, yielding good MAP acquired.

## VI. CONCLUSION

In this paper, we extend the traditional FCA (Formal Concept Analysis) technique into V-LCA (Viral Logical Concept Analysis) technique, in which each object and formal concept is featured by a logic formula. This formalism allows us to capture and present precisely behaviors of virus when constructing a concept hierarchy of malware. As results, we successfully generated a Malware Hierarchy from the dataset of 3000 real virus samples. Experiment results show that the concept hierarchy developed by our proposed V-LCA gained better quality than the traditional FCA.

## REFERENCES

[1] M. Igor, “Silicon implants,” *Virus Bulletin*, pp. 8-10, 1997.
[2] P. Szor, “Advanced code evolution techniques and computer virus generator kits,” *The Art of Computer Virus Research and Defense*, 2005.
[3] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, “Detecting malicious code by model checking,” in *Proc. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer Berlin Heidelberg, 2005, pp. 174-187.

[2] http://vxheaven.org/

[4] F. Song and T. Touili, "Efficient malware detection using model-checking," in *Proc. International Symposium on Formal Methods*, Springer Berlin Heidelberg, 2012, pp. 418-433.

[5] F. Song and T. Touili, "Pushdown model checking for malware detection," *International Journal on Software Tools for Technology Transfer 16*, no. 2, 2014.

[6] M. Huth and M. Ryan, "Logic in Computer Science: Modelling and reasoning about systems," Cambridge University Press, 2004.

[7] B. Ganter and Rudolf Wille, "Formal concept analysis: mathematical foundations," *Springer Science & Business Media*, 2012.

[8] G. Zhang, "Chu spaces, concept lattices, and domains," *Electronic Notes in Theoretical Computer Science 83*, pp. 287-302, 2013.

[9] B. Ganter, "Two basic algorithms in concept analysis," Springer, 2010.

[10] D. Poshyvanyk, M. Gethers and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2012.

[11] L. Wang, X. Liu and J. Cao, "A new algebraic structure for formal concept analysis," *Information Sciences*, vol. 180, no. 24, pp. 4865-4876, 2010.

[12] V. Duquenne, "Contextual implications between attributes and some representation properties for finite lattices," Springer, 2013.

[13] S. Obiedkov, "Modeling preferences over attribute sets in formal concept analysis," *Formal Concept Analysis*, pp. 227-243, Springer, 2012.

[14] R. Belohlavek and V. Vychodi, "Formal concept analysis and linguistic hedges," *International Journal of General Systems*, vol. 41, no. 5, pp. 503-532, 2012.

[15] L. Chaudron and N. Maille, "1st order logic formal concept analysis: from logic programming to theory," *Computer and Information Science*, 1998.

[16] B. Ganter and R. Wille, *Formal Concept Analysis – Mathematical Foundations*, Springer, 1999.

[17] S. Ferré and O. Ridoux, "A logical generalization of formal concept analysis," in *Proc. 8th International Conference on Conceptual Structures* in *Conceptual Structures: Logical, Linguistic, and Computational Issues,* 2000.

[18] D. Borchmann, F. Distel, and F. Kriegel, "Axiomatization of General Concept Inclusions from Finite Interpretations," *LTCS-Report 15-13, Chair for Automata Theory*. Institute for Theoretical Computer Science, Technische Universität Dresden, Dresden, Germany, 2015.

[19] A. Ketterlin, P. Gançarski, and J. Korczak, *Conceptual Clustering in Structured Databases: A Practical Approach*, 1995.

[20] T. Quan, S. Hui, and T. Cao, "A fuzzy FCA-based approach to conceptual clustering for automatic generation of concept hierarchy on uncertainty data," *CLA*, 2004.

[21] Y. Yue, T. Finley, F. Radlinski, and T. Joachims, "A support vector method for optimizing average precision," in *Proc. the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2007.

**Nguyen Thien Binh** is a Ph.D Student in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam. He received his B.Eng. degree in Information Technology from Post and Telecommunication Institute of Technology, Ho Chi Minh City, Vietnam, in 2004 and received his Master degree in 2010 from Bordeaux I University, France. His current research areas include formal methods, program analysis/verification, malware analysis.

**Tran Cong Doi** was born in Dong Nai Province, Viet Nam, in 1981. He received the B.E degree in electrical-electronics engineering and the M.E degree in computer science from Bach Khoa University, Ho Chi Minh City, Vietnam in 2009 and 2012. He has worked as a lecturer at Dong Nai University since 2012. His current research interests include: machine learning, formal concept analysis, artificial intelligence, and programming language.

**Quan Thanh Tho** is an associate professor in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam. He received his B.Eng. degree in Information Technology from HCMUT in 1998 and received Ph.D degree in 2006 from Nanyang Technological University, Singapore. His current research interests include formal methods, program analysis/verification, the Semantic Web, machine learning/data mining and intelligent systems. Currently, he heads the Department of Software Engineering of the Faculty. He is also serving as the chair of Computer Science Program (undergraduate level).

**Nguyen Minh Hai** is a Ph.D student in the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology (HCMUT), Vietnam. He received his B.Eng. degree in Information Technology from HCMUT in 2007 and received his Master degree in 2010 from Bordeaux I University, France. His current research areas include formal methods, program analysis/verification, malware analysis, security and dynamic scheduling. Currently, he is the lecturer in the Faculty of Information Technology, Industrial University of Ho Chi Minh city.