The Basic Building Blocks of Malware

Jinwook Shin and Diana F. Spears
University of Wyoming
Laramie, WY 82071
{jshin@, dspears@cs.}uwyo.edu

Abstract

Many security attacks accomplish their goals by controlling the inputs to a target program. Based on this insight, we have developed a novel and highly effective approach to developing malware signatures.[1] *These signatures, also called "basic building blocks" of malware, possess the essential elements common to all malware of a certain class. The key to the success of our approach is that it captures the global semantics of malware. Experimental evaluation shows that our algorithm can detect syntactic malware variants with low errors, and outperforms currently popular malware detection systems, such as McAfee VirusScan Enterprise.*

1 Introduction

The objective of our research is to detect malware, such as a virus, by recognizing its underlying goals. Rather than identifying and representing malware patterns syntactically, we adopt a semantic approach that discovers and captures the true underlying attack goal(s) of the malware. Why is a semantic approach preferable? First, different syntactic representations may have the same meaning. Second, it is easy for an attacker to obscure the program's real goals by inserting irrelevant function calls or changing the binary code's superficial appearance. Third, a significant portion of the malware might be incidental or irrelevant to its attack goal. For example, some code may merely perform normal operations, such as memory allocation and initialization, to set up for a subsequent real attack. In contrast, our approach detects all malware variants with *semantically* identical attack goals.

Christodorescu *et al.* [5] proposed the first approach to semantics-aware malware detection. However, their approach is manual and depends on local patterns; an attacker may easily mount her attack by avoiding such local attack patterns. To avoid this problem, we focus on *global*, rather than *local semantics*. Global semantics refers to the structure of the entire program as a whole, whereas local semantics refers to individual system calls. To the best of our

[1]Our B^3 approach has a patent pending. This project was supported by ONR URI grant "Anomaly and misuse detection in network traffic streams," subcontract PENUNV48725.

knowledge, our approach is the first to capture global semantics in a malware signature, along with almost complete automation.

Our malware signature, called a *basic building block* (b^3), is constructed by translating code to graphical structures, abstraction, extraction of semantics, and finally inductive inference. Our contributions are:

- **Globally semantic signatures**. Most prior malware signatures are syntactic; they consist of a short sequence of bytes that is unique to each attack. With these signatures, it is easy to bypass detection with minor binary changes to the program [4]. Even recent semantic signatures [5] find local patterns and are therefore easily vulnerable. We present a novel approach that focuses on attack goals using a globally semantic malware signature.
- **Automated signature construction.** A malware signature is normally generated manually by security experts, which typically takes hours or days whenever a new attack instance comes into the world. Our algorithm automatically generates the basic building blocks in a few minutes, depending on the number of training examples and program size.
- **Reduction in errors of omission (false negatives).** Unlike other modern malware detection systems, our algorithm is able to detect *unknown attack examples* with high probability. What makes this possible is that our algorithm is based upon a machine learning technique called *inductive inference*, which enables us to predict unknown examples. On the other hand, for certain classes of benign programs, our approach has an increase in the number of false positives (false alarms).
- **Wider applicability to modern attacks.** Our approach is applicable to any function-based malware program written in a high-level language. This is significant because most modern malware programs are in high-level languages.

2 Attack Goals

Initially, a malware program has no control over the target program. But it *does* have control over the *input* to the target program. It takes control of the target program via malicious input.

The input to the target results from malicious *outputs* from the malware program. We call the malware program's outputs its *attack goals.* We coarsely divide security attacks into *memory-based* (such as buffer overflow or format string attacks) and *function-call-based* (or simply *function-based*) attacks, according to the main strategy utilized by the malware program.

The focus here is on function-based attacks, which produce hostile *actions* on the victim's system, such as opening a TCP port to send a copy of itself to remote machines, dropping a backdoor, deleting or intercepting sensitive information, modifying system configurations of the victim's machine, and so on. The goal of function-based attacks is usually expressed as hostile actions, e.g., by invoking certain function calls. Most of today's viruses, worms, Trojans, backdoors, DoS tools, and other hacking tools, which are written in high-level languages such as C/C++, are function-based attacks.

The fundamental difference between previous research and our work in finding a malware signature is that we do not rely on local attack patterns, such as binary pattern matching. Instead, we take a global view as to what the ultimate attack goals are and how they relate to each other. To this end, we analyze outputs of a malware program – because they represent potential attack goals. For example, Figure 1 shows two example programs that are syntactically different, but have semantically identical goals. A typical modern malware detection system would create two signatures – one for each program. Our system recognizes their identical semantics and generates a single semantic malware signature.

As an aside, note that we are not claiming to have solved the program equivalence problem, which is undecidable. We have instead addressed the problem of determining whether a new unseen program belongs to a particular class of malware, which is a machine learning problem in the standard supervised learning paradigm [11].

3 B^3 Discovery Algorithm

3.1 Overview

A basic building block, which is a model of malware attacks, is constructed from a set of attack and non-attack programs as follows:

1. **Convert each program (attack or non-attack) into a graph.** A graphical representation is used because it is easier to generalize over. Since an attack program's source code is often unavailable, executable binary must first be transformed into a tractable high-level representation for the graph. The IDA Pro disassembler [6] is used to automate this process; it obtains assembly code from binary code. Because IDA Pro is unable to unpack/decrypt binary code, we first manually unpack and/or decrypt the program. The assembly code is then converted to a graph that is a hybrid of control flow and data dependence graphs.

2. **Partition the graph into subgraphs.** For abstraction, the overall graph is divided into subgraphs, each containing a program subgoal or terminal function.

3. **Semantic abstraction.** Semantic abstraction is the key to making our approach scalable. With abstraction, the graph is boiled down to its skeletal semantic essence. Our abstraction algorithm inputs a graph that has been divided into subgraphs, and outputs a finite-state machine (FSM) that captures global program semantics. An FSM representation has been chosen because it simplifies the induction process.

```
int main(void){
    FILE* fp = NULL;              //file pointer
    char* data = "abcde";
    fp = fopen("test", "w");      //opens a file
    if(fp == NULL) exit(1);       //if fopen() fails, then exit
    fputs(data, fp);              //writes data in the file
    fclose(fp);                   //closes the file

    CreateProcess(program1,...);  //runs a program

    int c;
    fp = fopen("foo", "r");       //opens a file
    c = fgetc(fp);                //reads data
    fclose(fp);                   //closes the file

    return 0;                     //returns to operating system
}
```

Program A

```
int main(void){
    HANDLE h;                     //file handle
    char  buffer[1024];
    strncpy(buffer, "abcde", 5);
    h = CreateFile("test"...);    //opens a file
    if(h = INVALID_HANDLE_VALUE)
                 ExitProcess(1);  //if CreateFile() fails, then exit
    WriteFile(h, buffer, 5,...);  //writes data in the file
    CloseHandle(h);               //closes the file

    WinExec("program1", SW_SHOW); //runs a program

    return 0;                     //returns to operating system
}
```

Program B

Figure 1: Two general programs in C. Programs A and B are syntactically different but have semantically identical goals ($goal_1$: write data into a file and $goal_2$: execute a process). Note that `fgetc` in program A does not contribute to emitting an output.

4. **Inductive inference.** The final step is to perform inductive inference (which is a form of machine learning) over *strings* (i.e., possible executions) of all the FSMs – for the purpose of inferring one general model (signature) of all malware seen so far that are in a certain class. With inductive inference, strings from attack FSMs are treated as "positive examples" and strings from non-attack FSMs as "negatives examples" to train on. After training on these examples, the general model will include features of attacks, while excluding features of non-attacks. The resulting general model is a basic building block, or b^3, of malware of a certain class. This b^3, which is in the form of a generalized string (i.e., a string with disjunction allowed), is used for classifying new, previously unseen programs as "ATTACK" or "NON-ATTACK."

Note that every step of this process has been fully automated, other than unpack/decryption. Each of these steps will now be described in detail.

3.2 Graph Construction and Pruning

Malware assembly code is converted to a graph. This graph is composed of both a control flow graph (CFG) and a data dependence graph (DDG). CFGs enable us to logically interconnect subgoals in the later abstraction phase, and DDGs help recover function call arguments, also used in abstraction.

DDGs are used to recover function call arguments. For each function call, our algorithm identifies a *function-call node* in the graph. The algorithm then follows reverse paths in the graph from the function-call node to its data sources in the data dependence graph. It halts when it gets to graph nodes containing values that are statically known, and it removes all subgraphs earlier than these nodes. This procedure, which is a form of backward slicing, significantly prunes the graph size. In static data flow analysis, some data values such as function pointers are impossible to recover because they are statically unknown. Each statically unknown value is replace with a question mark.

Backward slicing [7] is then performed with the CFG. The algorithm begins toward the end of the program, at the location where the program emits a malicious output intended to be sent to the target program as input. We predefine output-emitting functions and terminal functions to identify these locations in the program. The algorithm then follows the *reverse control flow edges* in the graph. During this backward CFG traversal, every subgraph identified as "not semantically critical" (i.e., not output-emitting in terms of security attacks) is pruned from the graph [15].

3.3 Subgraphs

After graph construction and pruning, the next step is to prepare for abstraction. The graph is divided into multiple subgraphs, called "subblocks." Each subblock will become an abstract element (indivisible unit/node) in an abstract graph, called a "finite-state machine/transducer." The fundamental basis of each subblock is either a *security-critical* or *terminal* function, to be defined next.

A *security-critical function* is one that generates a *suboutput*, i.e., an action that is critical from a security standpoint, and which can be used to formulate the final (attack) program's (malicious) output. Examples include creating/deleting a file, sending network data, or modifying system configurations. Any function that is not security-critical is called *non-security-critical.* A *terminal function* is one that causes a program to terminate, e.g., `exit`. An example of a *non-terminal function* is `send`.

The key to embedding semantics into our approach is that we divide security-critical and terminal functions according to their *semantic properties.* For example, functions such as `fputc`, `fputs` and `write` all share the same semantic functional meaning to the system: they write data to a file stream. A unique *function group number* is assigned to each group of semantically similar functions.

To detect subblock boundaries, a *semantic prologue (SP)* and *semantic epi-*

logue (SE) pair is defined. A semantic prologue is the set of functions that must be executed *before* a security-critical function call, and a semantic epilogue is the set of functions that must be executed *after* the function call.

In summary, a subblock consists of a security-critical or terminal function as its basis, and an SP-SE pair for subblock boundary delineation. For example, Figure 2 shows three subblocks for program A in Figure 1. Formally, we define a *subblock* as:

Definition 1 (Subblock) An attack graph (or, more generally, program graph) is defined as $G = \langle V, E \rangle$, where V is a set of vertices and E a set of edges. We define a function $distance(v_1, v_2)$ to output the number of edges on the shortest path from vertex v_1 to v_2. Let F be a set of security-critical functions and terminal functions, and $V_F \subseteq V$ be the set of graph nodes containing these functions. For each function $f \in F$, let $v_f \in V_F$ be the node that contains f. A *subblock* is a subgraph $G' = \langle V' \cup V'' \cup \{v_f\}, E' \cup E'' \rangle$ where

$V' = V'' = \{v_f\}$, $E' = \phi$ if and only if $SPE_f = \{\phi, \phi\}$. Otherwise,

- $V' = (V_{sp} \cup V_{spv}) \subseteq V$ and $V_{sp} = \{v\}$ and $v = argmin_{\vartheta \in SP_f}(distance(\vartheta, v_f))$, $V_{spv} = \{v' \mid v' \in V$ is on the shortest path from v to $v_f\}$, $E' = \{e \mid e \in E$ is on the shortest path from v to $v_f\}$.

- $V'' = (V_{se} \cup V_{sev}) \subseteq V$ and $V_{se} = \{w\}$ and $w = argmin_{\vartheta \in SE_f}(distance(v_f, \vartheta))$, $V_{sev} = \{v' \mid v' \in V$ is on the shortest path from v_f to $w\}$, $E'' = \{e \mid e \in E$ is on the shortest path from v_f to $w\}$.

3.4 Semantic Abstraction

The next step is to convert each graph into a form of finite-state machine called a *finite-state transducer (FST). The FST is an abstract graphical model of the global program semantics.* Each FST node (state) corresponds to a subblock. An FST is a type of finite-state machine whose OUTPUT is not just `ACCEPT` or `REJECT`; it is also a translator. Each transition in an FST is labeled with two symbols: `INPUT/OUTPUT`. Coinciding with the execution of each FST transition is the emission of the corresponding `OUTPUT` symbol.

`INPUT`s in the FST encode the subblocks, and `OUTPUT`s encode the subblock suboutputs. In particular, the function group number becomes the `INPUT` symbol, and the terminal or security-critical function arguments are translated to the `OUTPUT`, using a *translation function.* Recall that the function group number is a semantic notion; therefore, the INPUT symbol abstracts the semantic aspects of the attack. The reason for using function arguments for the `OUTPUT` is that a function's suboutput is strictly dependent upon its arguments. The function's suboutput can therefore be summarized semantically by specifying its arguments.

With this encoding scheme of `INPUT/OUTPUT` symbols, we can now formally define the new data structure to which the pruned and subdivided (into subgraphs) graph is converted. This new data structure is called an *abstract-FST.*

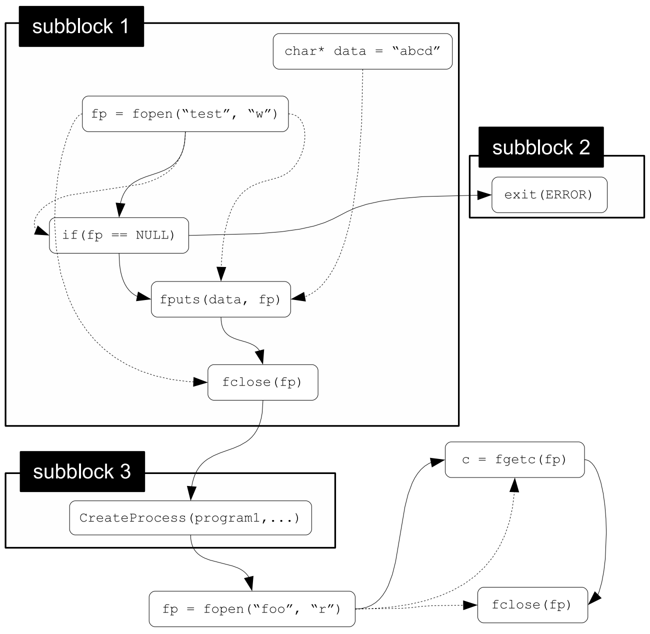

Figure 2: Subblocks of program A in Figure 1. Dotted lines indicate data dependencies and solid lines control flows. Note that there are only three subblocks in the graph because `fgetc` is a non-security-critical function.

An abstract-FST is a very concise graphical representation of the global semantic essence of the original (attack) program:

Definition 2 (Abstract-FST) $A = (\Sigma, Q, q, F, \Gamma, \delta)$, where:

- Σ: a finite set of INPUT symbols
- Q: a finite set of states (i.e., subblocks),
- $q \in Q$: the initial state (which is the starting subblock in the program),
- $F \subseteq Q$: the finite set of final states (subblocks),
- Γ: a finite set of OUTPUT symbols, and
- δ: the transition function (edges), which is defined as $\delta : Q \times \Sigma \rightarrow Q \times \Gamma$.

Note that some subblocks may be both security-critical and terminal. If this is the case, we split the subblock into two subblocks (a security-critical subblock and a terminal subblock) and serially connect them. If there is a control branch from a subblock, an empty subblock is inserted at the branching point, with ε-transitions as connections. For example, Figure 3 shows the abstract-FST for the programs A and B in Figure 1. Note that they both translate semantically to the same abstract-FST.

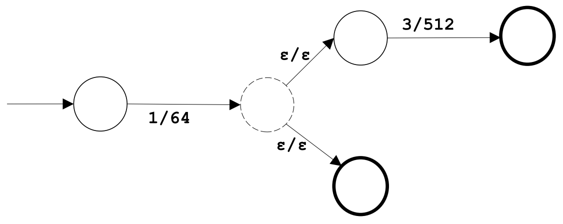

Figure 3: Abstract-FST for programs A and B in Figure 1. Bold circles represent the final subblocks. Since subblock 3 is both security-critical and terminal, an extra final subblock is appended at the end of it. An empty subblock (the second subblock from the left) is inserted for a control branch.

3.5 Inductive Inference

This section describes the basic building blocks of malware that are inferred from a set of attack and non-attack programs, which have been converted (as described above) into abstract-FSTs. *Concept learning* is used to infer a model (or *hypothesis*) from the FSTs. This model, which is a basic building block (b^3) of malware, can be used to classify future examples (programs) as either `ATTACK` or `NON-ATTACK`. Concept learning is a form of inductive inference, or simply "induction" [11].

The inductive process consists of five steps. It is assumed that abstract-FSTs have been formed from every attack or non-attack program, as described in the previous sections. For the first step, we take each abstract-FST, and extract all possible *strings* from it.[2] A string is a single execution sequence of an FST that begins in an initial state of the FST, follows the FST transitions, and terminates in a final state of the FST. Figure 4 (at the top) shows two example FST strings. If an FST is derived from an attack program, then its extracted strings are labeled *attack strings*; if it is derived from a non-attack program, then its strings are labeled *non-attack strings*. For the second step, every attack and non-attack string is augmented with a frequency vector. We call an augmented attack string a *positive example*, and we call an augmented non-attack string a *negative example*. The third step consists of aligning examples, in preparation for learning (training). After alignment, learning consists of two steps: generalization (step 4) to incorporate all positive examples into the model, followed by specialization (step 5) to exclude all negative examples from the model. Inductive inference generalizes the model by taking the union of it and all positive examples, and then specializes the model by taking the difference between the model and all negative examples. The result is a model that captures the commonalities of attacks, and omits features of non-attacks. Each of these steps will be described in more detail in the following subsections.

[2]If there is a loop in an FST, the loop is followed only once. This creates loop-free strings.

3.5.1 Creation and Alignment of Examples

Concept learning is done over positive and negative examples, which are collectively called *training examples* for the learner.

To convert a string, x, to a positive or negative training example, it is necessary to augment the string with a *frequency vector*, V. The purpose of this vector is to give higher weight to more frequent attack patterns. In particular, the frequency vector encodes information regarding which INPUT symbols appear more often in the positive examples and less often in the negative examples. It is initialized to be all 0s, and it is updated during induction (as described in the following subsection).

Prior to induction, examples are aligned according to their INPUT symbols. We use a sequence alignment technique from [12] to find an optimal alignment between strings (and, later, between a string and the model) and to calculate a *similarity score*. These scores are divided by the maximum string (sequence) length – to express similarity as a percentage. All strings that are aligned are made to have the same length. An underscore (_) sign denotes a placeholder, which gives strings the same length. When strings are aligned, we often see this placeholder along with an ϵ-transition. Note that $|x|$ is defined to be the length of string x, where ϵ's are included.

3.5.2 Induction Over Training Examples

Our induction algorithm consists of two phases: generalization and specialization. *Generalization* finds the commonalities among the positive examples via a *union operator*. It does this one example at a time. In other words, it begins by taking the union of two positive examples to create an initial model. Then it continues to take the union of the model with the remainder of the positive examples, thereby continuing to generalize the model.

Recall that a b^3 is a model of malware attacks. To understand the form of a b^3, it is necessary to formally define a model. A model is similar to an example, except it includes disjunction. In other words, a model can combine multiple alternative execution paths in one data structure. Formally,

Definition 3 (Model) A *model*, $m = (\Sigma, Q, S, f, \Gamma, \delta, V)$, is a sub-machine of an abstract-FST with additional information attached. Here, S is a set of initial states, f is a final state, and V is the *frequency vector*. Note that the length of model m, i.e., $|m|$, is equal to the maximum length of any training example.

There is only one terminal subblock for any sub-machine and this terminal subblock does not emit any output. Recall that if a subblock is both output-emitting and terminal, then we split the subblock into two subblocks and serially connect them, so that the last subblock is always terminal, not output-emitting. Therefore, when we align two sub-machines, the terminal subblocks are always merged into a single final state.

Next, we define the union operator that performs generalization. To simplify our formal definitions of the union and difference operators, training examples are expressed using the same notation as models. In fact, they are actually degenerate models (i.e., models without disjunction), so this is reasonable.

Definition 4 (Union Operator) Assume two aligned examples, $a_1 = (\Sigma_1, Q_1, S_1, f, \Gamma_1, \delta_1, V_1)$ and $a_2 = (\Sigma_2, Q_2, S_2, f, \Gamma_2, \delta_2, V_2)$, or an aligned model, a_1, and an example, a_2. The *union operator* ($\cup$) on a_1 and a_2 is defined to be $a_1 \cup a_2 = (\Sigma_1 \cup \Sigma_2, Q_1 \cup Q_2 \cup Q_e, S_1 \cup S_2 \cup S_e, f, \Gamma_1 \cup \Gamma_2 \cup \Gamma, \delta, V)$, where the transition function $\delta = (Q_1 \cup Q_2) \times (\Sigma_1 \cup \Sigma_2) \rightarrow (Q_1 \cup Q_2) \times F_g(\Gamma_1 \times \Gamma_2)$, and F_g is an implementation-specific function defined as $F_g : \Gamma_1 \times \Gamma_2 \rightarrow \Gamma$. In other words, F_g is a function that computes a set of new OUTPUT symbols Γ from Γ_1 and Γ_2. Q_e and S_e are defined below.

The union operation merges equivalent aligned states in the following manner. Let us define any pair of states $q_1 \in a_1$ and $q_2 \in a_2$, that are aligned, and for which the INPUTs (on the outgoing edge) are equal to be *equivalent*. These equivalent states merge into a single state q in $a_1 \cup a_2$. Any transitions leading into/out of q_1 or q_2 in a_1 or a_2 now lead into/out of this single state q in $a_1 \cup a_2$. The OUTPUT of this state becomes a function of the product of the OUTPUTs of the two original states, i.e., $\gamma = F_g(\gamma_1, \gamma_2)$, where $\gamma_1 \in \Gamma_1$, $\gamma_2 \in \Gamma_2$, and $\gamma \in \Gamma$, and $\Gamma = F_g(\Gamma_1 \times \Gamma_2)$. Also, a state with ϵ INPUT is considered equivalent to any state aligned with it during the union (but not difference) operation. Its INPUT becomes that of the other state with which it is aligned. If a ϵ OUTPUT symbol appears in any of q_1 or q_2 or both, then F_g returns either ϵ or an implementation-specific value. Finally, Q_e is the set of all aligned states that are equivalent in a_1 and a_2, e.g., $q \in Q_e$. S_e is the subset of these that are start states.

During the union operation, the frequency vector V is updated with the following sequence of steps:

1. $V = \mathbf{0}$
2. $V = V_1 + V_2$.
3. If the nth INPUT symbols in a_1 and a_2 match, then increase the nth element in V by one.

After generalization has completed over all positive examples, specialization is performed over all negative examples. Specialization subtracts each negative example, one-by-one, from the model via a *difference operator* – to omit elements specific to non-attacks.

Definition 5 (Difference Operator) Assume a model, $a_1 = (\Sigma_1, Q_1, S_1, f, \Gamma_1, \delta_1, V_1)$ and a negative example, $a_2 = (\Sigma_2, Q_2, S_2, f, \Gamma_2, \delta_2, V_2)$, that are aligned. Let Q_e be the set of all aligned states that are equivalent (see above) in a_1 and a_2. S_e is the subset of these that are initial states. The *difference operator* ($-$) on a_1 and a_2 is defined to be $a_1 - a_2 = (\Sigma_1, Q_1 - Q_e, S_1 - S_e, f, \Gamma_1, \delta, V)$, where $\delta = \delta_1$ with the exception that any transition to a deleted state goes instead to the successor of the deleted state.

V is updated with the following rule:

1. $V = V_1$.
2. If the nth INPUT symbols in a_1 and a_2 match, then remove the nth element in V.

The idea of deleting the nth element is not to give any weight to the subblock that exits in non-attacks. Figure 4 gives an example of generalization followed by specialization.

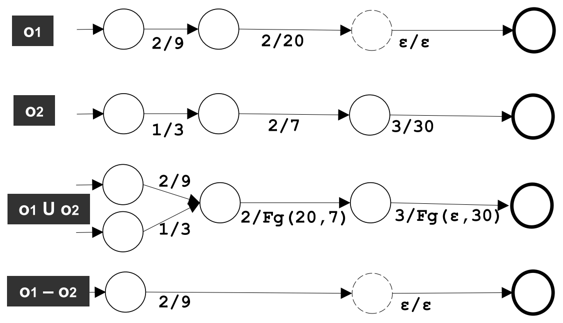

Figure 4: Two aligned examples a_1 and a_2 become an initial model ($a_1 \cup a_2$) via generalization and a refined model ($a_1 - a_2$) via specialization.

The output of this generalization-specialization process is a b^3, which is an attack model. This b^3 can be used to classify new, unseen examples (see Section 4).

3.5.3 Parameter Optimization and Final b^3

Our classification algorithm (Section 4, below) uses partial matching between the b^3 and an example, for flexibility. In particular, the algorithm calculates a maximum matching threshold, k. To be labeled an attack, the similarity score of a new example must exceed k.

To compute k, the model and an example are aligned according to INPUT symbols, and both INPUT symbols and *decoded* OUTPUT symbols are used to calculate the matching score. We first compare corresponding INPUT symbols, and if they match then we compare OUTPUT symbols. The decoded OUTPUT symbol is a list of positive integers and which one to use is implementation-specific (i.e., we only use the value with the highest weight). For the OUTPUT symbol comparison, we use another parameter, β, to allow matches within some range ($\pm\beta$). Therefore the matching score k is dependent upon β.

The value k is also dependent on another parameter called the *subgoal window*, γ, which tolerates a partial rather than total match between the order of subgoals in the model and new example. Finally, k is weighted to compute the final matching score, α. We use the frequency vector, V, to give greater weight to matches with more frequent subgoals.

A separate b^3, with this parameter optimization, is constructed for each attack group (class):

Definition 6 (Basic Building Block) A *basic building block*, $b^3 = (\Sigma, Q, S, f, \Gamma, \delta, V, \alpha, \beta, \gamma)$, is the final attack model that has been formed from generalization, specialization, and parameter optimization over *all* training examples.

The sub-machine at the bottom in Figure 4 is a very simple (for illustration purposes) example of a basic building block.

4 Classification

The following approach is used to classify new unseen examples as "ATTACK" or "NON-ATTACK." Each new example is compared with the b^3. Recall that partial matching is used. To calculate a similarity score (or matching score) between the learned b^3 and previously unseen examples, we do the following. First, we obtain β and γ for the learned b^3 and use those parameters to compute a new similarity score α' between the b^3 and the unseen submachines. This score is used to classify new examples. A new example is only labeled an "ATTACK" if α' exceeds the threshold α from the learned b^3.

5 Experimental Results

We tested our algorithm against all variants of 23 attack groups (see Table 1). For each group, we divided the attack variants into two subgroups for training (i.e., to construct a b^3) and testing (i.e., to test the b^3's classification accuracy on unseen test examples). We performed induction using the attack training examples plus 120 randomly-chosen benign programs from a fresh Windows installation. For all attack groups, we tried a token translator, length translator, and character distribution translator for translation functions. A token translator extracts character strings in the arguments, a length translator encodes the argument length in bytes, and a character distribution translator encodes the character distribution in the arguments.

Attack Type	Attack Group Name
Worm	Donghe, Vorgon, Deborm, Klez, Libertine, Nimda, Gizer, Energy, Kelino, Shorm
Virus	CIH, Emotion, Belod, Evul, Mooder, Team, Inrar, Eva, Lash, Resur, Spit
Hacking Tool	Auha
DoS Tool	Lanxue

Table 1: Attack groups for training and testing.

After the learning phase, our algorithm was evaluated on the testing examples of the aforementioned attack groups. While testing, we excluded any examples that could not be processed by IDA Pro. Since we assume that disassembly can be performed successfully by IDA Pro before detection, we do not take into account those failed examples. Out of 79 testing examples, our algorithm missed one instance of Auha but detected the rest of the attack variants in the testing group (98.73% detection). We also tested the b^3s against 1032 randomly-chosen normal programs. The system detected 2 normal programs (`telnet.exe`, `wupdmgr.exe`) as attack (0.19% false positive).

In order to see if our algorithm is resilient to minor binary changes, we generated the basic building blocks from randomly chosen CIH samples from [2] and tested against the original copy of CIH.1010b and CIH.2690 and the signature-removed version of CIH.1010b and CIH.2690. *Our algorithm successfully detected all of them.* Also, we tested McAfee [1] VirusScan Enterprise ver. 8.0.0 with the latest virus definition against CIH.1010b and CIH.2690 virus samples obtained from [2]. VirusScan successfully detected the original copies but it failed to detect them after we manually removed (zeroed-out) the CIH.1010b and CIH.2690 signature from the virus body with a binary editor.

One of the disadvantages of our approach is that our system may classify benign programs as attacks if the benign programs are semantically similar to attacks. This is a possible explanation for the two mis-classified examples (`telnet.exe`, `wupdmgr.exe`).

Malware detection must be efficient. Table 2 shows the low average CPU time and memory taken to transform unknown programs to examples and then classify them as attack or non-attack.[3]

Target Size (KB)	4∼40	4∼100	100∼400	400∼1024
Time (sec)	52	210	288	381

Table 2: Average CPU time taken for classification.

6 Related Work

There are two complementary approaches to malware detection: *static* and *dynamic*, each approach having both strengths and weaknesses. This paper focuses on a static approach. One very effective and popular static approach is that of Sung *et al.*, called SAVE [18]. Their malware signature is derived from an API calling sequence and they mapped each API to an integer number to encode the sequence. They used a sequence alignment algorithm to compute a similarity score to compare malware variants. However the resulting API sequence is nothing more than a piece of syntactic information – therefore an adversary may create another malware variant to defeat the system in such a way that the program has a totally different API sequence but still has the same semantic attack goal. Furthermore, an attacker can randomize the API

[3]These times were taken using an Intel Pentium M 1.0GHz CPU with 512MB memory.

sequences by inserting arbitrary APIs in the middle of the sequence that do not affect the original attack goal.

In response to the problems with syntactic signatures, there has been a very recent but growing trend toward semantic malware detection. A handful of publications on the topic have appeared in the last couple of years. In 2005, Christodorescu *et al.* [5] were the first researchers to provide a formal semantics for malware detection. They manually developed a template that describes malware semantic properties and demonstrated that their algorithm can detect all variants of certain malware using the template with no false positives. Wang *et al.* [20] proposed a system called *Shield*, which has vulnerability-specific, exploit-generic network filters for preventing exploits against security vulnerabilities. Shield is resilient to polymorphic or metamorphic worms. Sokolsky *et al.* [17] used bisimulation to capture some of the semantics, but not at an abstract level. Bruschi *et al.* [3] invented a semantic approach to handling automated obfuscations. Kinder *et al.* [9] used model checking to semantically identify malware that deviates from a temporal logic correctness specification. Scheirer and Chuah [14] developed a semantics-aware NIDS to detect buffer overflow exploits.

There are two major reasons why our approach presents an advance beyond these prior approaches. First, other than the model checker, all of these other approaches look for local, rather than global, semantic attack patterns. Second, they all require significant manual intervention, e.g., to develop a template, graph, or other data structure representing desirable (or attack) behavior. The problem with using local attack patterns is that an attacker can at any time take advantage of this fact and mount her attack by avoiding the local attack patterns in her program. Furthermore, by capturing global semantics, a regular grammar (rather than a context-free grammar as needed by Wagner and Dean [19]) suffices for signatures. This results in a substantial computational advantage. The problem with manual intervention is that it is time-consuming and impractical. In contrast to these prior semantic approaches, ours looks for global patterns and is almost fully automated.

Some researchers have focused on automating the generation of attack signatures, e.g., Autograph [8], Honeycomb [10], and EarlyBird [16] analyze network streams to automatically produce signatures by extracting common byte patterns and they are used to detect unknown Internet worms. Unfortunately, these approaches are syntactic and local.

The most relevant prior work to our inductive inference approach is that of Kephart *et al.*, who developed a statistical method for automatically extracting signatures from a corpus of machine code viruses [13]. Their approach differs from ours because it is syntactic which, as mentioned above, is problematic.

7 Summary and Future Work

We have presented a basic building block discovery algorithm to detect malware variants. Our approach is globally semantics-aware and automated. Although our approach cannot handle some of the more challenging malware, such as code that self-mutates at run-time, or is specially packed/encrypted or obfuscated, it is nevertheless broadly applicable. In particular, experimental evaluation has demonstrated that our algorithm can detect a wide variety of unknown attacks (viruses, worms) with low errors, and it is resilient to minor binary changes.

Future work will focus primarily on optimizing the speed of our approach, further testing over more examples, and methods for recovery after a malware attack has been identified by a b^3.

Acknowledgement

The authors would like to thank Insup Lee for suggesting the problem of identifying the basic building blocks of malware.

References

[1] Mcafee - antivirus software and intrusion prevention solutions. `http://www.mcafee.com/`, Last accessed on 10 Nov. 2005.

[2] Vx heavens. `http://vx.netlux.org/`, Last accessed on 10 Nov. 2005.

[3] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment.* IEEE Computer Society, 2006.

[4] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 34–44, New York, NY, USA, 2004. ACM Press.

[5] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[6] DataRescue. Ida pro - interactive disassembler. `http://www.datarescue.com/idabase`, Last accessed on 10 Oct. 2005.

[7] S. Horwitz, T. Reps, and F. Binkley. Interprocedural slicing using dependence graphs. *Transactions on Programming Languages and Systems*, 12(1), 1990.

[8] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[9] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Lecture Notes in Computer Science 3548*, pages 174–187. Springer Verlag, 2005.

[10] Christian Kreibich and Jon Crowcroft. Honeycomb - Creating intrusion detection signatures using honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II)*, Boston, November 2003.

[11] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[12] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[13] Jeffrey O.Kephart and William C.Arnold. Automatic extraction of computer virus signatures. pages 178–184. 4th Virus Bulletin International Conference, 1994.

[14] Walter Scheirer and Mooi Chuah. Network intrusion detection with semantics-aware capability. In *Proceedings of the Second International Conference on Security and Systems in Networks*. IEEE Computer Society, 2006.

[15] Jinwook Shin. The basic building blocks of attacks. Master's thesis, University of Wyoming, 2006.

[16] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.

[17] Oleg Sokolsky, Sampath Kannan, and Insup Lee. Simulation-based graph similarity. In *Lecture Notes in Computer Science 3920*. Springer Verlag, 2006.

[18] Andrew H. Sung, Jianyun Xu, Patrick Chavez, and Srinivas Mukkamala. Static analyzer of vicious executables (save). In *ACSAC*, pages 326–334, 2004.

[19] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, Washington, DC, USA, 2001. IEEE Computer Society.

[20] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204, New York, NY, USA, 2004. ACM Press.