2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis

Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, Dongyan Xu
Department of Computer Science and CERIAS, Purdue University
West Lafayette, IN, USA, 47907-2107
{gu16, kpei, wang868, lsi, xyzhang, dxu}@cs.purdue.edu

***Abstract*—Currently cyberinfrastructures are facing increasingly stealthy attacks that implant malicious payloads under the cover of benign programs. Existing attack detection approaches based on statistical learning methods may generate misleading decision boundaries when processing noisy data with such a mixture of benign and malicious behaviors. On the other hand, attack detection based on formal program analysis may lack completeness or adaptivity when modeling attack behaviors. In light of these limitations, we have developed LEAPS, an attack detection system based on supervised statistical learning to classify benign and malicious system events. Furthermore, we leverage control flow graphs inferred from the system event logs to enable automatic pruning of the training data, which leads to a more accurate classification model when applied to the testing data. Our extensive evaluation shows that, compared with pure statistical learning models, LEAPS achieves consistently higher accuracy when detecting real-world *camouflaged attacks* with benign program cover-up.**

***Keywords*—Attack Detection; Statistical Learning; Program Analysis;**

I. Introduction

Enterprise cyberinfrastructures are facing more severe cyber threats powered by sophisticated attack techniques. Such attacks are driven by financial interests for divulging privacy records, collecting competitor's intelligence, or concealing unauthorized system accesses. They may exploit system vulnerabilities or leverage social engineering (i.e., psychological manipulation of innocent people to perform harmful operations unintentionally) to initiate attacks, leaving only inconspicuous footprints. More recently, instead of only launching one-time attacks, adversaries tend to implant stealthy and persistent backdoors — which parasitize in the memory space of some long-running benign applications or embed in the application's binaries — to facilitate future security penetrations. Based on the cloaking properties of such attacks, in this paper we call them *camouflaged attacks*.

Recent research efforts on host-based attack detection can be divided into two categories: program analysis based methods and statistical learning based methods.

Attack Detection Based on Program Analysis: Some approaches [1]–[5] perform static analysis on applications (assuming the availability of source or executable code) to obtain precise program execution models. But the non-trivial overhead, complexity of accurate binary analysis, and intentional obfuscation limit their applicability to real-world applications/environments. Other detection systems [6]–[8] perform dynamic analysis in a training phase and build deterministic program behavior models by profiling application-system interactions.

Attack Detection Based on Statistical Learning: Instead of achieving precise program models like in the former category, detection systems in this category utilize statistical learning techniques to build benign/malicious classification models. For example, in the work of [9], [10], association and frequency rules are learned from training data for future detection. In other systems [11], [12], histogram-based methods are applied to profiling normal program behavior. A more sophisticated hidden Markov model (HMM) is adopted in [13], [14] for intrusion detection. More recently, the works in [15]–[18] utilize Support Vector Machine (SVM) to build binary classification models. One major advantage of these statistical learning based systems is that they are robust in dealing with incomplete data, and thus can usually achieve better classification results compared with program analysis based approaches.

We argue that, for the detection of *camouflaged attacks*, current attack detection systems may encounter difficulties in effectively discriminating between benign and malicious behavior. The main reason is that the extraction of pure malicious behavior in a raw dataset (e.g., system execution logs) is difficult. For trojaned applications or runtime application exploitations belonging to *camouflaged attacks*, the malicious payload no longer executes independently. Instead, it runs concurrently with the benign code of the application, which generates a training dataset with interleaved benign and malicious behaviors. Such noisy training datasets may eventually lead to a biased classification boundary.

In light of the limitation above, we have developed LEAPS[1] to integrate the capabilities of the two camps. LEAPS is inspired by a recently proposed vision called "Learn-2-Reason" [19], which promotes mutual enhancement between statistical learning and formal analysis methods. Specifically, LEAPS leverages program execution analysis to refine its statistical learning model, boosting its detection accuracy.

Taking a host-based system event log as input, we adopt the supervised statistical learning model to classify benign and malicious events. The classification model is built upon system-level features extracted from the log, such as system event, libraries, and functions. The effectiveness of this approach is based on the key observation that the system-level behavior of anomalous execution, triggered by the malicious code, is different from the system-level behavior of benign code.

[1]LEAPS stands for **L**earning **E**nhanced with **A**nalysis of **P**rogram **S**upport

978-1-4799-8629-3/15 $31.00 © 2015 IEEE
DOI 10.1109/DSN.2015.34

Then, to address the noisy training dataset problem in detecting *camouflaged attacks*, we use the control flow graph (CFG) of each benign application (which may not be complete) as the oracle to guide the training. From our observation, benign and malicious instructions by nature cluster separately in the memory space. For each data point in the noisy training dataset, we measure its distance to the benign CFG and assign a corresponding weight, which indicates that outlying data points are more likely to be events triggered by the malicious payload. Although injecting malicious code near benign code is not impossible (e.g., injecting malicious code in free alignment areas between procedures), it is usually not used in real-world attacks because such limited space greatly restricts the functionality of injected code. Typical attacks choose to allocate extra memory for malicious payloads and then hijack benign control flows.

Taking the assigned weights into consideration, we build a Weighted Support Vector Machine (WSVM) classifier to detect benign and malicious behaviors. Deriving a complete and accurate CFG using static analysis on a binary is a well-known challenge due to binary obfuscation and software protection mechanism. In LEAPS, we avoid static program analysis by dynamically inferring the CFG of each application — based on the stack walk trace in the system event log. We note that such a CFG is by no means complete, but it presents a general execution pattern of the application, which is sufficient for our distance approximation.

This paper makes the following contributions:

- A better statistical learning model for detecting *camouflaged attacks*, guided by CFGs derived from program trace analysis. This model is especially suitable for noisy training datasets mixed with benign/malicious events.
- An algorithm for CFG inference only based on the stack walk trace in the system event log, without requiring static program analysis or program instrumentation.
- Extensive evaluation of LEAPS for the detection of *camouflaged attacks* with diverse combinations of applications, malicious payloads, and attack methods, demonstrating effectiveness of LEAPS.

We organize the rest of this paper as follows. Section II presents the threat model and the overview of the workflow. Section III provides the system design of LEAPS and Section IV presents implementation details. Section V shows extensive evaluation of LEAPS in different attack scenarios. Section VI discusses current limitations and proposes future work. Section VII describes related work and we conclude in Section VIII.

II. System Overview

In this section, we first discuss the threat model and the attacks we target. Then we present the general workflow of LEAPS and give a brief introduction of the functionality of each component.

A. Threat Model

We assume that the adversaries have already found a way to infiltrate the system. They may achieve this through physical access to a target computer, e.g., manually replacing an application with a trojaned version, or using some social engineering techniques to trick innocent users to click some malicious web sites or open a disguised attachment in a phishing email. They may also remotely exploit some unpatched vulnerabilities and then implant a backdoor into some long-running benign program. We do not intend to use LEAPS to raise an alarm at the time of intrusion, instead we aim to detect the anomalous behavior and backtrack to its entry point when the remote adversary performs malicious actions through the persistent backdoor implanted in the system.

In this paper we focus on *camouflaged attacks*, which run under the cover of some benign program. This is a common technique to make malicious behavior more difficult to detect. Finally, we require that system event logging function be turned on so that it can generate program execution traces as input to our analysis.

B. Workflow of LEAPS

Similar to traditional anomaly detection systems, we divide the workflow of LEAPS into two phases: *Training Phase* and *Testing Phase*.

1) Training Phase: We illustrate the workflow of the *Training Phase* in Figure 1. Here we give a brief description of each component and its input data format.

The initial input data consist of raw log files generated by the system event logging engine. Event logging systems are commonly equipped in modern operating systems for diagnosing application performance problems, thus they are able to walk the application stacks to backtrack execution when system events are captured. These raw system event log files are recorded in a controlled environment and will be used as training data. The *benign raw log* is generated when we execute a clean version of an application; whereas the *mixed raw log* is generated when the parasitic malicious payload (embedded in the binary or injected through remote exploitation) and the benign application code run in the same process context, leading to interleaved execution of benign and malicious code.

Our *Raw Log Parser* is similar to the front end of Introperf [20]. We parse the raw log file, correlate stack walk traces with corresponding system events, and extract function and library information sliced for each process in both the user and kernel space. The output, which we term *stack-event correlated log*, consists of itemized system events for the application of interest. Moreover, each event is attached with its stack walk trace annotated with libraries and functions.

The *Stack Partition Module* is for splitting the stack walk trace of each event into two parts, *application stack trace* and *system stack trace*. *Application stack trace* consists of the stack walk within the application itself. We use this to infer the application's CFG because it contains both explicit and implicit execution information. *System stack trace* consists of stack walk trace in the shared libraries and the operating

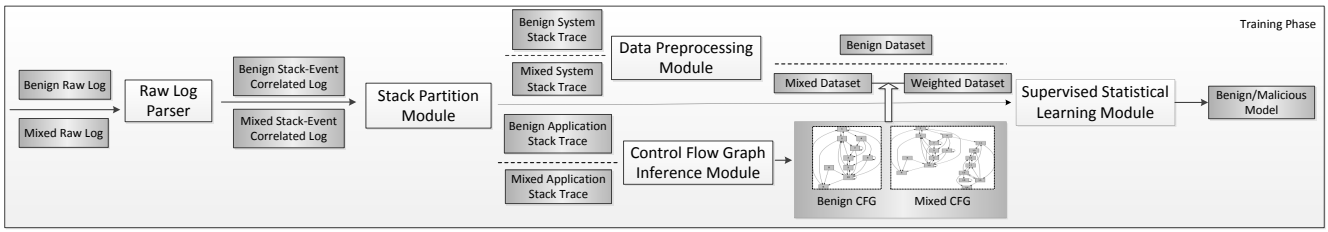

Fig. 1: Workflow of the *Training Phase* of LEAPS

system (OS) kernel. We note that the differences in system-level behavior (e.g., system events, shared libraries, and library/kernel functions) are best suited for distinguishing the benign functionality from the malicious functionality. Thus we extract features used by the statistical learning model from the *system stack trace* in the system event log.

The *Data Preprocessing Module* extracts features from the *system stack trace*. Here we apply hierarchical clustering to group the functions and libraries into clusters. This generates both the *benign dataset* and the *mixed dataset*, which are ready to be used by the statistical machine learning engine.

The *Control Flow Graph Inference Module* builds the CFG of the application by inspecting the *application stack trace*. We construct two CFGs separately from the *benign application stack trace* and the *mixed application stack trace*. Then we compare these two CFGs to measure the distance of each execution path in the mixed CFG to the benign CFG. As we can map each execution path to its affiliated system event, we assign a weight (computed based on the distances of all execution paths attached to this event) to each event in the *mixed dataset* (generated by the *Data Preprocessing Module*) and generate a *weighted dataset*.

Our *Supervised Statistical Learning Module* is a unified learning system for building the benign/malicious classifier. We employ a Weighted Support Vector Machine, which is a binary classification model, to obtain the classifier based on the training data generated from the *Data Preprocessing Module*. We treat the data in the *benign dataset* as the positive samples in the statistical learning model, while the data in the *weighted mixed dataset* are viewed as the negative samples. We can apply the learned benign/malicious classifier to detect attacks from production system logs.

2) Testing Phase: After the *Training Phase*, we have generated application-wise binary classifiers from the training data. In the *Testing Phase*, first we perform application slicing on the system event log (same as in the *Training Phase*) to generate the testing data. Then we apply the classification models (targeting different application/payload combinations) to the testing data for detection.

We point out that we use the application-wise binary classifier only for the convenience of evaluation. When applied to attack detection in real situations, LEAPS can coalesce all application data from the system event log to learn a universal classifier for testing.

III. System Design

Following the workflow in the previous section, we now highlight some key techniques we have developed for LEAPS and describe the algorithms behind them.

A. Data Preprocessing

Data preprocessing is the essential step before applying any statistical learning model. It requires domain knowledge to interpret the raw data, extract distinguishing features for classification, and discretize these features to be ready as the input to statistical learning. Because the statistical learning model is general and not specific to our raw data, data preprocessing is critical to the effectiveness of the final classification model generated.

In LEAPS, we choose to use system events and information in their correlated system-level stack traces to characterize the program behavior being executed. As mentioned in Section II, after parsing the raw log file, we are able to correlate the stack walk traces with their corresponding system events. Stack walk trace entries contain the function invocations leading to this event from the application. Then we partition the stack trace and only select the *system stack trace* and system events as input to the data preprocessing module.

Each entry in the *system stack trace* contains both the library and function information. We aggregate the libraries and functions of each event and generate a 3-tuple entry: $\{Event\_Type, Lib, Func\}$. $Event\_Type$ stands for the type of this system event. Lib and $Func$ stand for the set of libraries and functions in the *system stack trace* of this event. $Event\_Type$ is well defined in the system, and thus can be naturally mapped to the integer space. For Lib and $Func$, we leverage hierarchical clustering [21] to group similar library/function sets into one cluster. We use set dissimilarity as the metric to calculate a pairwise distance matrix, DM, as follows:

$$\mathrm{DM[i][j]} = \mathrm{set\_dissimilarity(i,j)} = 1 - \frac{\|\mathrm{set_i} \cap \mathrm{set_j}\|}{\|\mathrm{set_i} \cup \mathrm{set_j}\|} \quad (1)$$

We utilize this pairwise distance matrix in the hierarchical clustering model to obtain optimal clusters. Finally we replace Lib and $Func$ in the 3-tuple entry with its corresponding cluster number and $Event\_Type$ with the integer based on its event type. Figure 2 gives a concrete example of preprocessing a SysCallEnter event and its 3-tuple entry result. We use these discretized 3-tuple entries as the input data to the statistical learning model.

B. Control Flow Graph Inference

In our approach, we need the CFG of the benign application execution as an oracle to process the log mixed with the benign and malicious execution. While CFGs of binary executables can be acquired using static or dynamic analysis, generating CFGs statically from binaries is challenging due to various

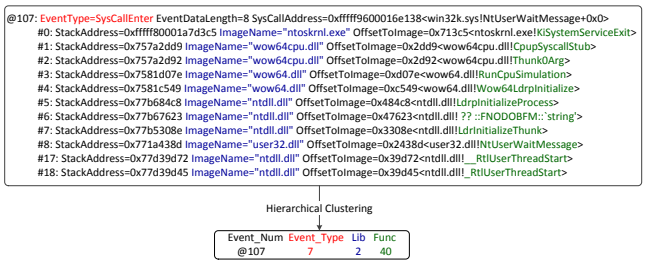

Fig. 2: The Result of Conducting Hierarchical Clustering on a System Event

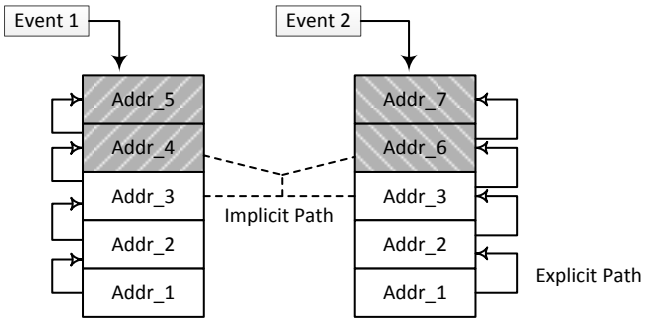

Fig. 3: Example of Control Flow Graph Inference

difficulties such as identifying function boundaries [22], distinguishing instructions from data entries, dynamic loaded libraries, obfuscation, binary packing, and the impracticality of instrumenting real world binaries to collect fine grained dynamic execution information. Hence, we decide to derive CFGs only from the *application stack trace* extracted from the system event log. While the completeness of the inferred CFG is dependent on the frequency of the system events and the exercised functionality when logging is enabled, it is sufficient to produce an incomplete CFG that can approximately reflect the general execution structure of the application. As we will show later, we leverage a heuristic algorithm to predict the missing parts of the benign CFGs and recognize malicious payloads that do not belong to the original benign graphs. Therefore, a unique advantage of LEAPS is that it only relies on the system log, without analyzing the binaries.

We give a concrete example in Figure 3. For each individual event, there is an *application stack trace* attached to it. There are two events shown in this figure. For *Event 1*, the *application stack trace* starts from $Addr\_1$ to $Addr\_5$. *Event 2* is the subsequent event and its stack trace becomes different from *Event 1* after $Addr\_3$, which invokes $Addr\_6$ and $Addr\_7$. We are able to identify two types of control flow within the *application stack trace*. We call the first type of control flow an *explicit path*, which indicates the function invocations in the stack trace. For example, the execution path from $Addr\_1$ to $Addr\_2$ is an *explicit path*. We call the other type of control flow an *implicit path*, which we infer from stack traces of two adjacent events. In Figure 3, $Addr\_3$ invokes $Addr\_4$ in *Event 1* and $Addr\_6$ in *Event 2*, which indicates there is a control flow from $Addr\_4$ to $Addr\_6$ in the program. Based on these two criteria, we build the CFG incrementally by enumerating all events and their *application stack traces*.

We present the detailed algorithm in Algorithm 1. In Line 12, we find the branch point by comparing two adjacent stack

Algorithm 1 Control Flow Graph Inference

```
Input:  funcentry ← GEN_CFG
        ast ← stack_trace_file
        cfg ← empty_dict
1:  procedure ADDTO_CFG(cfg, start, end)
2:      if cfg.haskey(start) then
3:          cfg[start].add(end)
4:      else
5:          cfg[start] := set([end])
6:  procedure BRANCH_POINT(prev_stacklist, curr_stacklist)
7:      index := COMMON_PREFIX_LEN(prev_stacklist, curr_stacklist)
8:      return index
9:  procedure GEN_CFG(ast, cfg)
10:     while line do
11:         if isEvent(line) then
12:             branchidx := BRANCH_POINT(prev_stacklist, curr_stacklist)
13:             ADDTO_CFG(cfg, prev_stacklist[branchidx], curr_stacklist[branchidx])
14:             for i ∈ [0, LEN(stacklist)-1] do
15:                 ADDTO_CFG(cfg, curr_stacklist[i], curr_stacklist[i+1])
16:             prev_stacklist := curr_stacklist
17:             curr_stacklist.clear()
18:         else if isStack(line) then
19:             funcaddr := EXTRACT_FUNCADDR(line)
20:             curr_stacklist.push(funcaddr)
21:         line := ast.readline()
```

traces and add the *implicit path* in Line 13. In Line 15, we add the *explicit paths* for all the function invocations within one stack trace.

We apply this CFG inference algorithm on both the *benign application stack trace* and the *mixed application stack trace*. Thus we are able to generate two CFGs. Figure 4-(1) shows the CFG of a benign execution of Vim, whereas Figure 4-(2) shows the CFG of a trojaned Vim that contains the malicious payload of a Reverse TCP Shell. By comparing these two CFGs (e.g., aligning nodes with the same address in two graphs), it is not difficult to identify that the left subgraph of the *Vim mixed CFG* is similar to the *Vim benign CFG* because both use the benign functionality of Vim. But the right subgraph of the *Vim mixed CFG* is unique, indicating that this is more likely to be from the anomalous execution caused by the malicious payload. We point out that, although the CFG alone may be used as a attack signature for detection, it is not robust enough when encountering polymorphic malware in the real world. This is the reason we introduce the statistical learning model for a behavior-based attack detection system.

C. Weight Assessment

With the inferred *benign CFG*, we aim to assess the degree of "benignity" for each event in the *mixed dataset*. We show the algorithm for the weight assessment in Algorithm 2.

The input to this algorithm is the *benign CFG* and the *mixed CFG* inferred from the *application stack traces*. When building the CFG from the *mixed application stack trace*, we also create a reverse mapping, named *memap* in Algorithm 2's input, from the program path to the event number.

We start by iterating each program path in the *mixed CFG*. We check whether the start and end vertices of this path are also connected in the *benign CFG*. If they are connected, we assign 1 to the weight (whose range is $[0, 1]$) for this path. Otherwise, it means this path does not exist in the *benign CFG*.

As mentioned before, the inferred CFG is not complete. It is possible that some paths in the *mixed CFG* are benign,

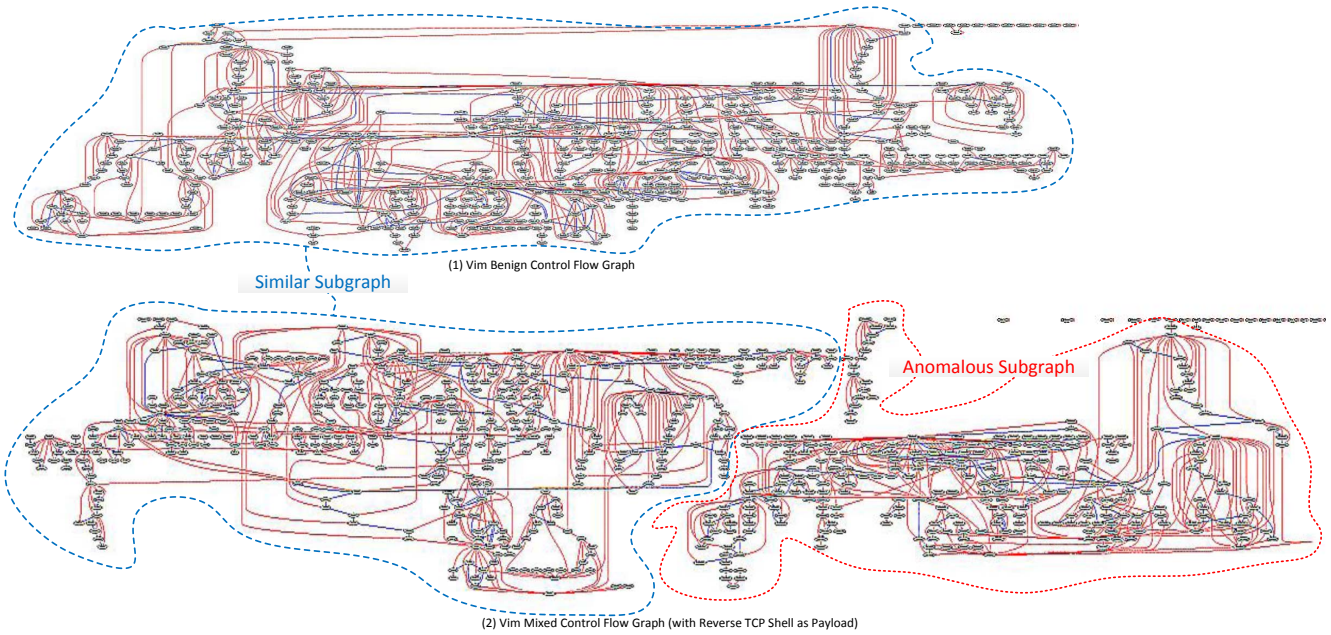

Fig. 4: Comparison of (1) Vim Benign CFG and (2) Vim Mixed CFG

Algorithm 2 Weight Assessment

```
Input:  funcentry ← COMPARE_CFG
        bcfg ← benign_cfg_dict
        mcfg ← mixed_cfg_dict
        memap ← mixed_event_dict
1:  procedure GEN_CFG_DENSITY(cfg)
2:      for start, endset ∈ cfg.iter() do
3:          for end ∈ endset do
4:              density_array.add(start)
5:              density_array.add(end)
6:      return SORT(density_array)
7:  procedure CHECK_CFG(start, end, cfg, level)
8:      if start = end ∧ level ≠ 0 then
9:          return True
10:     valueset := cfg.get(start)
11:     if valueset.empty() then
12:         return False
13:     level := level + 1
14:     for value ∈ valueset do
15:         if CHECK_CFG(value, end, cfg, level) then
16:             return True
17:     return False
18: procedure SET_WEIGHT(eventmap, key, weight, result)
19:     if eventmap[key] ≠ nil then:
20:         for eventnum ∈ eventmap[key] do
21:             if result[eventnum] = nil then
22:                 result[eventnum] := {'weight':weight, 'number':1}
23:             else
24:                 number := result[eventnum]['number']
25:                 result[eventnum] := {REBALANCE(weight,number), number+1}
26: procedure ESTIMATE_WEIGHT(addr, density_array)
27:     addr_idx := BISECT(density_array, addr)
28:     mindiff := MIN(start - density_array[addr_idx-1], density_array[addr_idx] - addr)
29:     weight := 1 - mindiff/(density_array[addr_idx] - density_array[addr_idx-1])
30:     return weight
31: procedure COMPARE_CFG(bcfg, mcfg, memap)
32:     density_array := GEN_CFG_DENSITY(bcfg)
33:     for start, endset ∈ mcfg.iter() do
34:         for end ∈ endset do
35:             if CHECK_CFG(start, end, bcfg) then
36:                 weight := 1
37:             else
38:                 if WITHIN_RANGE(start, end, density_array) then
39:                     weight := ESTIMATE_WEIGHT(start, density_array)
40:                 else
41:                     weight := 0
42:             SET_WEIGHT(memap, start+end, weight, result)
```

but missing in the *benign CFG* due to its incompleteness. For example, some additional benign functionality might be executed and recorded in the *mixed system log*, but not in the *benign system log*. In order to address this problem, we create a *density array* by inserting all the addresses of nodes appearing in the *benign CFG*. For any path that is not in the *benign CFG*, if it is in the range of this *density array*, we estimate its weight based on its normalized distance to the closest nodes in the *benign CFG*. For all other paths that exceed the boundary of the *density array*, we assign 0 as its weight. This weight assessment approach is based on the observation that code close to the benign code is more likely to be benign and code far away from the benign code is more likely to be malicious. That is also the reason why LEAPS can tolerate the incompleteness of the inferred CFG.

With the weight for each program path in the *mixed CFG*, we search the reverse mapping *memap* to find its corresponding event number. Each event may have multiple paths mapped. We compute the weight of each event by averaging all its paths' weights.

D. Binary Classification Model

The building of the benign/malicious classification model is a key component in LEAPS. Given the *benign dataset* and *mixed dataset* with assigned weights, our goal is to learn an accurate binary classifier from these training data. This classifier will be used to distinguish malicious events from benign ones in the unseen testing data.

We build two binary classification models for comparison. The first is purely based on the system-level function call graph (with no statistical learning) and the second uses WSVM. We discuss their strengths and weaknesses separately and compare the results quantitatively in the evaluation section.

1) Decision Model Based on System-level Call Graph: System-level behaviors, such as functions from shared libraries

and the OS kernel, represent the interactions between applications and their underlying execution environment. These features are widely adopted in anomaly detection systems to reveal aberrant execution of the application. Conceptually similar to existing system behavior based classification systems [23], [24], we build our first classification model based on the system-level function call graph (built from the *system stack trace* in the system event log). From the *benign/mixed system stack trace*, we extract the function invocation chain from the stack trace of each event. Thus we can build the two system-level function call graphs, the *benign call graph* (BCG) and *mixed call graph* (MCG), separately. We use the former as the positive model and the latter as the negative model. In the *Testing Phase*, we extract the call relations from the stack trace in the testing data and check them in both the *BCG* and *MCG*. We make a classification decision for each individual event based on the existence of such call relations in both call graphs.

From the results presented in Section V, we find that the hit rates are low for classifying benign testing data for all datasets. The first reason for this is that the system-level call graph model is not able to classify data points that do not appear in the training set. The second reason is that the stack traces of benign events may exist in both the *BCG* and *MCG*, which make it difficult for the model to accurately predict their classes.

Furthermore, we find that in some specific datasets (e.g., chrome_reverse_https and chrome_reverse_tcp), the hit rates are also low for classifying malicious testing data. We manually check the events that lead to this problem. The main cause is that some outlier points may greatly affect the classification decision. To give a specific example, consider a kernel function invoked by both benign and malicious code. Assume the benign code only calls the function once and the malicious code calls it 1000 times, the corresponding function invocation edges in both call graphs will be the same. Thus the call graphs cannot yield any information of the invocation frequency. Further assume that this function invocation appears in the testing data. From a statistical perspective, this invocation is likely to be from the malicious code, but it will be classified as "undecidable" by the call graph model.

2) Weighted Support Vector Machine: Considering the limitations of the call graph model above, we design a more sophisticated binary classification model based on statistical learning. There are multiple machine learning techniques for learning binary classifiers, such as Logistic Regression (LR) [25], SVM [26], and Decision Tree [27]. Due to the discriminative classification power of SVM and its popularity, we use SVM to build our classification model. Furthermore, to incorporate the weights assigned to the training data, we employ a Weighted SVM method in this work to find an optimal classifier by taking the confidence of each data point into consideration.

Suppose there are n training data points from both the benign and mixed datasets, denoted $D = \{(x_i, y_i, c_i), i = 1, \ldots, n\}$ where x_i is the feature generated from data preprocessing for the $i\text{-}th$ data point and y_i is its binary label. We treat the benign data as positive samples, while the mixed data are viewed as negative samples, i.e., $y_i = 1$ for the benign data and $y_i = -1$ for the mixed data. c_i corresponds to the weight.

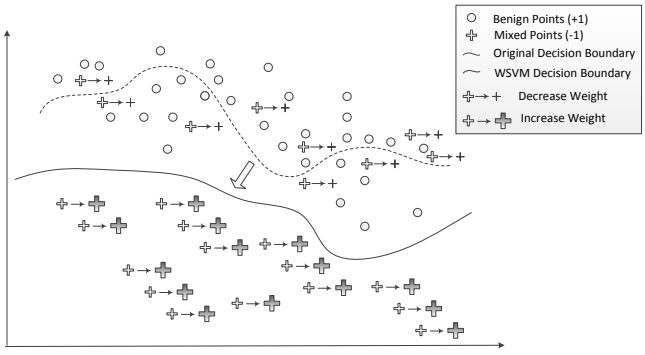

Fig. 5: An illustration of the classifiers learned by the original SVM model and the Weighted SVM model.

Note that the weight c_i is a real value between 0 and 1. For the benign data, the weight is simply 1. For the mixed data, we obtain the weight from the weight assessment in Section III-C. The purpose of the Weighted SVM is to learn a classifier w, which can accurately distinguish benign data from malicious data. We give the formulation of the Weighted SVM as follows:

$$\begin{aligned} \min_{w,\xi} \quad & \|w\|^2 + \lambda \sum_i c_i \xi_i \\ s.t. \quad & y_i w^T x_i \geq 1 - \xi_i \\ & \xi_i \geq 0 \end{aligned} \tag{2}$$

here ξ_i is the classification error of the $i\text{-}th$ data point. $w^T x_i$ is the prediction score of x_i based on the classifier w, i.e., the larger the value, the more likely x_i is benign. The term $\sum_i c_i \xi_i$ in the objective function is the total classification loss/error weighted by the importance of the data, which we are trying to minimize. The term $\|w\|^2$ is the regularizer on the classifier to avoid the overfitting problem [25], which is widely adopted in statistical machine learning applications. λ is the trade-off parameter to balance the two terms. The constraint enforces that the prediction of the data point, $w^T x_i$, is consistent with its label y_i. For example, for a benign point with label $y_i = 1$, if the classifier's output, $w^T x_i$, is negative, then the model will incur a large classification error ξ_i due to this constraint.

Based on the generalized representer theorem [28], the minimizer to the optimization problem in Eqn. 2 exists and has a representation of the form:

$$w^T x_i = \sum_{j=1}^{n} \alpha_j y_j k(x_i, x_j) \tag{3}$$

where $k(x_i, x_j)$ is a kernel function defined on the feature space. We use a Gaussian Kernel, $k(x_i, x_j) = \exp(-\frac{\|x_i - x_j\|^2}{\sigma^2})$, in this work, and σ^2 is the radius parameter. Substituting Eqn. 3 into Eqn. 2, we can obtain an equivalent problem:

$$\begin{aligned} \min_{\alpha} \quad & -\sum_i \alpha_i + \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j k(x_i, x_j) \\ & s.t. \quad 0 \leq \alpha_i \leq \lambda c_i \end{aligned} \tag{4}$$

The above optimization problem can be solved efficiently using a quadratic programming solver. By minimizing this objective function, we can achieve an optimal classifier. We illustrate the difference between the original SVM method and the

Weighted SVM method in Figure 5. We can see from this figure that the classifier learned from the original SVM model may misclassify benign points to malicious. The reason is that a certain amount of mixed data points actually belong to benign events. By minimizing the classification error on these mislabeled data points, the SVM classifier does not perform well especially on benign data. On the other hand, by assigning proper weights via CFG guidance (i.e., decreasing the weights of mislabeled points and increasing the weights of true malicious points), the classifier learned from the Weighted SVM distinguishes the benign points from the malicious ones more accurately.

In the *Testing Phase*, we apply the learned classification model to the testing data x_t to give the prediction as follows:

$$y_t = w^T x_t = \sum_{i=1}^{n} \alpha_i y_i k(x_i, x_t) \quad (5)$$

where x_t is classified as malicious if $y_t < 0$.

IV. Implementation

We leverage the Event Tracing for Windows (ETW) [29] framework to log system events and generate stack walk traces. The ETW framework is a general-purpose tracing engine equipped in the latest Windows operating systems (first introduced in Windows 2000). It provides a tracing mechanism to log events triggered in multiple system layers, from user applications to kernel components. ETW has been widely adopted by third-party management tools for performance diagnostics. The output of ETW is an Event Tracing Log (ETL) file, which is the raw input to LEAPS. ETW allows us to enable stack walking for a selection of system events, e.g., system call, process/thread creation, image load/unload, file operations, registry tracing, etc. We parse the raw ETL file to generate a *stack-event correlated log*. We perform all ETW logging on a machine with an Intel Core i7 3.40 GHz CPU, 12GB RAM, and Windows Server 2008 R2 64-bit operating system.

We implement the *Stack Partition Module*, *Data Preprocessing Module*, and *Control Flow Graph Inference Module* in Python. When grouping the library and function set in the *Data Preprocessing Module*, we use the hierarchical clustering implementation in the clustering package of SciPy[2] and choose UPGMA method as the linkage criterion, i.e., the distance between any two clusters is the mean distance between all elements of each cluster.

We implement the *Supervised Statistical Learning Module* under the LIBSVM [26] framework. LIBSVM[3] is an integrated system for support vector classification, regression and distribution estimation with a wide range of machine learning applications. The input of the Weighted SVM model is the benign and mixed (with weights) training data. The output of LIBSVM is a binary classification model, which we use for attack detection in our testing data. In our implementation, we use 10-fold cross validation [25] to tune the model parameter λ and σ^2 on the training set.

[2]http://docs.scipy.org/doc/scipy/reference/cluster.hierarchy.html

[3]http://www.csie.ntu.edu.tw/~cjlin/libsvm/

V. Evaluation

In this section, we report our evaluation results on the effectiveness of LEAPS. First we describe the datasets in our experiments, i.e., the source of the data and the criteria of data selection. Then we discuss the procedure of our experiments and the measurements of the evaluation. Finally we examine three representative cases in detail and present the results of all other cases briefly.

A. Dataset

1) Data Source: We use 21 datasets (Table I) of different combinations of applications, malicious payloads, and attack methods to evaluate our approach. We categorize the attack methods into two groups: *offline infection* (malicious payload embedded in a benign binary) and *online injection* (malicious payload injected into a benign process at runtime). Each dataset consists of three subsets: a) *pure benign samples*, b) *mixed samples*, and c) *pure malicious samples*.

We obtain *pure benign samples* by exercising the benign application. *Mixed samples* are from profiling either trojaned applications (i.e., offline infection) or tampered processes (i.e., online injection). Thus *mixed samples* contain both benign and malicious events. *Pure benign samples* and *mixed samples* can be naturally collected in the real environment. We use them as positive/negative samples for training. As we mentioned before, because *mixed samples* contain benign events as noise, classifiers learned by traditional statistical learning methods are not accurate.

Pure malicious samples are difficult to obtain in a real environment because malicious payloads are always attached to benign applications. For this evaluation, we manually extract the malicious payloads and recompile them as independent malware. Here we only use *pure malicious samples* as the **ground truth for testing** to verify the effectiveness of our binary classifier on negative samples. After hierarchical clustering, each subset contains three features: *Event_Type*, *Lib*, and *Func*.

2) Data Selection: We select the training data for learning a binary classifier from: a) *pure benign samples* (positive training samples) and b) *mixed samples* (negative training samples). We select the testing data from: a) *pure benign samples* (positive testing samples) and c) *pure malicious samples* (negative testing samples). To avoid training and testing on the same benign samples, we divide the *pure benign samples* into two non-overlapping parts, 50% for training and 50% for testing. Taking the order of adjacent events into account, we increase the dimensions from 3 up to 30 by coalescing each 10 consecutive samples into one 30-dimension data point. Due to the large number of data samples, we randomly select 20% of the samples from each dataset to form the training and testing sets. In this way, we can achieve reasonable running time for the training phase and also near-complete coverage of the behavior in each dataset.

B. Evaluation Procedure and Measurement of Effectiveness

We compare our CFG guided Weighted SVM approach (denoted WSVM in Figure 6 and 7) with the other two classification approaches, i.e., approaches based on the system-level

TABLE I: Evaluation Results of LEAPS on Camouflaged Attacks

Name	Attack Method	Application	Payload	ACC	PPV	TPR	TNR	NPV
winscp_reverse_tcp	Offline Infection	WinSCP	Reverse TCP Shell	0.932	0.999	0.865	0.999	0.881
winscp_reverse_https	Offline Infection	WinSCP	Reverse HTTPS Shell	0.927	0.991	0.862	0.992	0.878
chrome_reverse_tcp	Offline Infection	Chrome	Reverse TCP Shell	0.877	0.998	0.755	0.999	0.803
chrome_reverse_https	Offline Infection	Chrome	Reverse HTTPS Shell	0.907	0.998	0.815	0.999	0.844
notepad++_reverse_tcp	Offline Infection	Notepad++	Reverse TCP Shell	0.846	0.998	0.693	0.998	0.765
notepad++_reverse_https	Offline Infection	Notepad++	Reverse HTTPS Shell	0.866	0.998	0.733	0.998	0.789
putty_reverse_tcp	Offline Infection	Putty	Reverse TCP Shell	0.886	0.815	0.998	0.774	0.998
putty_reverse_https	Offline Infection	Putty	Reverse HTTPS Shell	0.869	0.999	0.739	0.999	0.793
vim_reverse_tcp	Offline Infection	Vim	Reverse TCP Shell	0.914	0.995	0.832	0.996	0.856
vim_reverse_https	Offline Infection	Vim	Reverse HTTPS Shell	0.919	0.998	0.839	0.999	0.861
vim_codeinject	Offline Infection	Vim	Pwddlg	0.852	0.985	0.715	0.989	0.776
notepad++_codeinject	Offline Infection	Notepad++	Pwddlg	0.802	0.948	0.639	0.965	0.728
putty_codeinject	Offline Infection	Putty	Pwddlg	0.802	0.919	0.661	0.942	0.736
putty_reverse_tcp_online	Online Injection	Putty	Reverse TCP Shell	0.894	0.825	0.999	0.789	0.999
putty_reverse_https_online	Online Injection	Putty	Reverse HTTPS Shell	0.869	0.999	0.738	0.999	0.792
notepad++_reverse_tcp_online	Online Injection	Notepad++	Reverse TCP Shell	0.927	0.991	0.861	0.992	0.877
notepad++_reverse_https_online	Online Injection	Notepad++	Reverse HTTPS Shell	0.845	0.998	0.690	0.999	0.763
vim_reverse_tcp_online	Online Injection	Vim	Reverse TCP Shell	0.963	0.933	0.998	0.928	0.998
vim_reverse_https_online	Online Injection	Vim	Reverse HTTPS Shell	0.919	0.995	0.842	0.996	0.863
winscp_reverse_tcp_online	Online Injection	WinSCP	Reverse TCP Shell	0.950	0.996	0.904	0.996	0.912
winscp_reverse_https_online	Online Injection	WinSCP	Reverse HTTPS Shell	0.921	0.998	0.843	0.998	0.864

call graph (denoted CGraph in Figure 6 and 7) and traditional SVM, on all 21 datasets. We set the model parameters λ and σ^2 using 10-fold cross validation on the training set. To eliminate fluctuation caused by the random selection of the training and testing sets, we average all results over 10 runs.

We measure the performance of the classification results based on: *True Positives* (TP), *True Negatives* (TN), *False Positives* (FP), and *False Negatives* (FN). TP indicates actual benign samples that are correctly classified as benign. Similarly, TN represents malicious samples that are correctly classified as malicious. FP indicates malicious samples that are misclassified as benign. FN represents benign samples that are misclassified as malicious. Based on these four results, we evaluate the performance of the different methods by five measurements: 1) *Accuracy* (ACC), 2) *Positive Predictive Value* (PPV or Precision), 3) *True Positive Rate* (TPR or Recall), 4) *True Negative Rate* (TNR or Specificity), and 5) *Negative Predictive Value* (NPV) [30].

1) Accuracy: By definition, the *ACC* is the portion of the true results (both TP and TN) in the total test samples.

$$ACC = \frac{TP + TN}{TP + FP + FN + TN} \tag{6}$$

According to Figure 6 and 7, the *ACCs* of all applications elevate by varying degrees when using WSVM compared to SVM and CGraph. For example, the *ACC* of *winscp_reverse_https_online* increases from 59.9% (CGraph) to 92.1% (WSVM), which reflects a significant improvement on the overall hit rate of both benign and malicious prediction.

Though *ACC* indicates the overall performance of a binary classification, it may yield misleading results if the data set is unbalanced. Thus, we introduce four other measurements based on the confusion matrix (TP, TN, FP, FN) to give a more comprehensive evaluation of the experimental results.

2) Positive Predictive Value: Also known as *precision*, PPV measures the portion of actual benign samples in all predicted benign samples.

$$PPV = \frac{TP}{FP + TP} \tag{7}$$

As we can see from Figure 6 and 7, WSVM produces the highest *PPV* values. For instance, the *PPV*s of *putty_reverse_tcp_online* are 71.2% (CGraph), 79.6% (SVM) and 82.5% (WSVM).

3) True Positive Rate: Also known as *recall*, *TPR* measures the number of instances that are correctly classified as benign out of the total benign instances.

$$TPR = \frac{TP}{TP + FN} \tag{8}$$

The *TPR* of WSVM has obvious improvement for all 21 cases. For example, in Figure 7, the *TPR* of *putty_reverse_https_online* increases from 41.7% (CGraph) to 56.4% (SVM) and reaches 73.8% (WSVM).

4) True Negative Rate: True Negative Rate is also known as *specificity*. Similar to *TPR*, *TNR* calculates, out of the instances that are actually malicious, the number of instances that are correctly classified as malicious.

$$TNR = \frac{TN}{FP + TN} \tag{9}$$

From Figure 6, the *TNR* of *vim_codeinject* increases from 67.9% (CGraph) to 98.9%(WSVM). We have similar improvements of *TNR* on all other 20 cases according to Figure 6 and 7.

5) Negative Predictive Value: Similar to *PPV*, *NPV* measures the portion of the actually malicious samples out of the total predicted malicious samples.

$$NPV = \frac{TN}{TN + FN} \tag{10}$$

Again, WSVM ranks the highest in all 21 applications in terms of *NPV*. For instance, the *NPV* of *putty_reverse_https_online* increases from 69.9% (SVM) to 79.2% (WSVM), as seen in Figure 7.

C. Results and Discussion

Figure 6 and 7 show the results of the offline infection and online injection datasets respectively. We also present the detailed results of all datasets in Table I. From these figures, we can see that the proposed CFG guided Weighted SVM method achieves the best results on all measurements in all cases. In the rest of this section, we discuss three representative cases in detail.

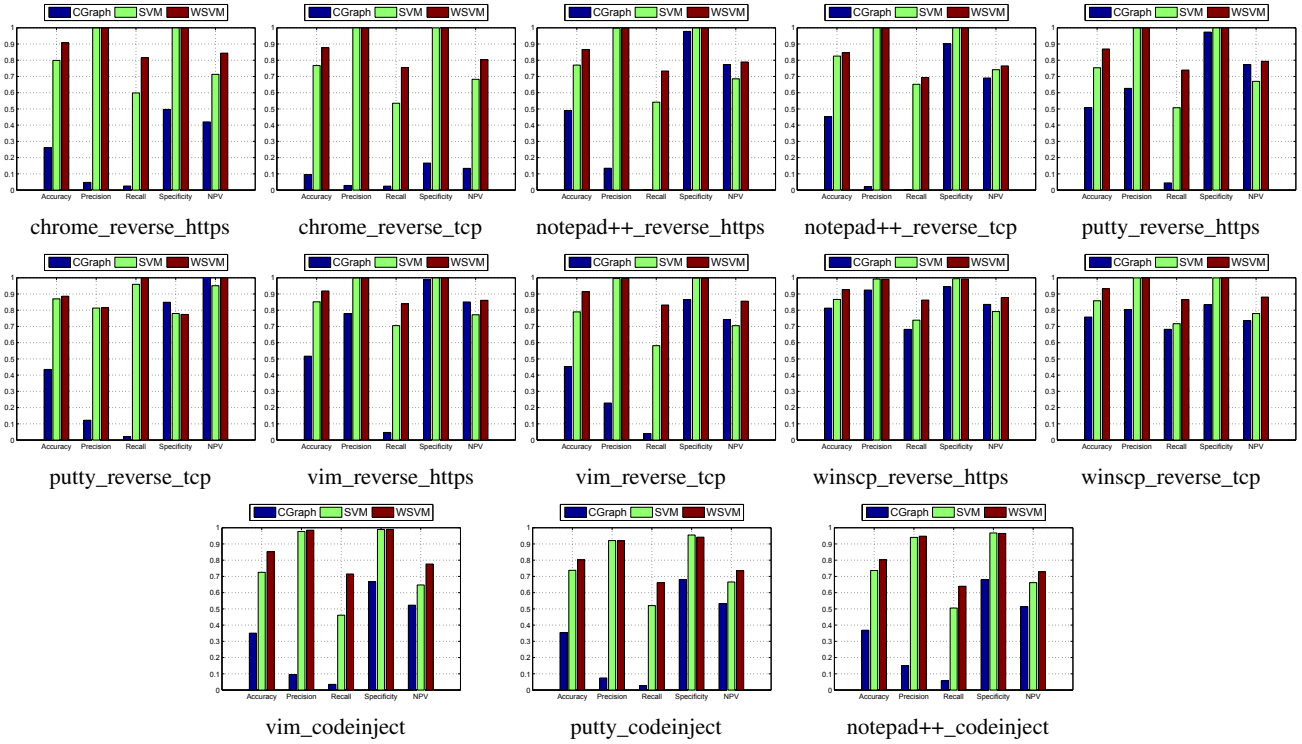

Fig. 6: Results Comparing LEAPS (WSVM) with System-level Call Graph and SVM for Offline Infection Detection

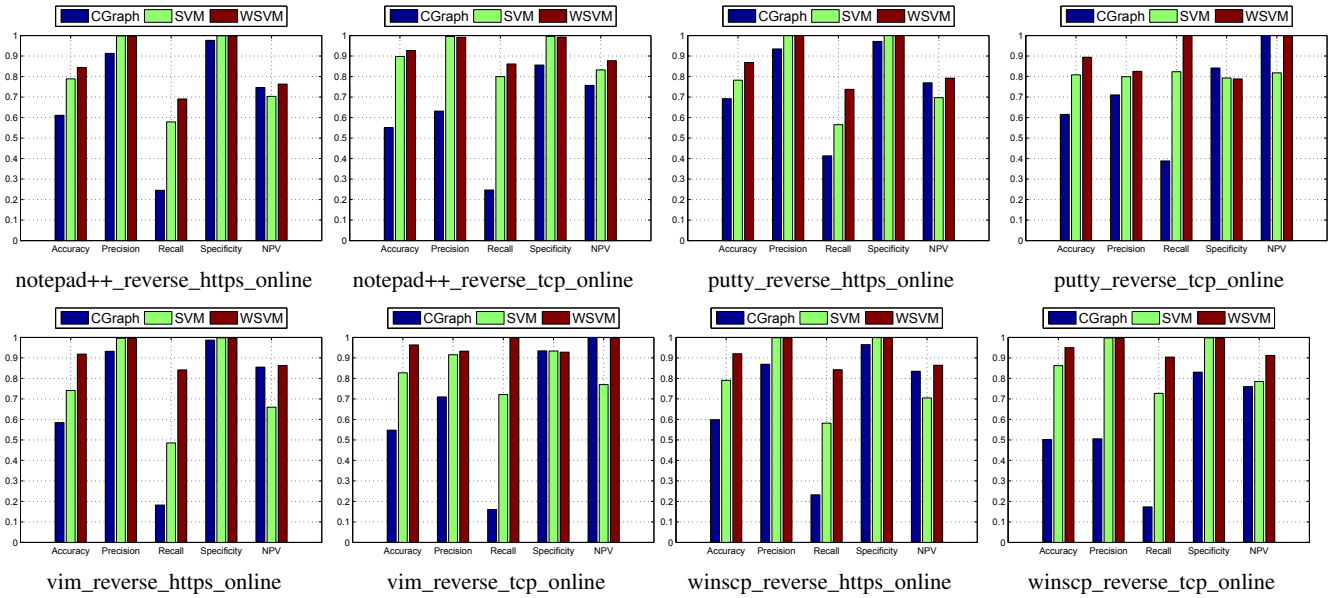

Fig. 7: Results Comparing LEAPS (WSVM) with System-level Call Graph and SVM for Online Injection Detection

1) Case Study I — winscp_reverse_tcp: This attack is in the offline infection category. The adversary can choose an arbitrary benign application binary and transform it into a trojaned application. They can implant the malicious payload into the binary and detour the program path at some specific location to trigger the payload. The payload may create a persistent backdoor and wait for a command from the remote adversary. After the adversary creates the backdoor, the trojaned program returns back to the normal control flow of the benign application. Legitimate users cannot detect that the remote hacker has already controlled their machine when the trojaned application is running.

In this case, we leverage the tools and payloads in the Metasploit Framework [31] to generate the trojaned application. Metasploit Framework is a widely-adopted system for developing and executing exploit code to perform penetration testing. Msfpayload is a command-line tool for generating

different types of shellcode in the Metasploit Framework. We use this tool to generate a Meterpreter, a dynamically extensible payload that uses in-memory DLL injection stagers. The Meterpreter communicates with the remote server via a reverse TCP connection. It enables the remote adversary to perform all kinds of hacking operations on a victim system, e.g., keylogging, file uploading, taking screenshots, password hash collection, etc. The benign host application in this case is WinSCP. We leverage Msfencode to encode the payload with shikata_ga_nai (a polymorphic XOR additive feedback encoder) three times and then embed it into the WinSCP binary.

We can see from the results that all five measurements increase if we use the WSVM model. Take *ACC* and *TPR* for instance, we can see from Figure 6 that these two measurements based on the call graph model are 74.79% and 68.16%. *ACC* and *TPR* increase to 85.81% and 72.08% if we use traditional non-weighted SVM. Our Weighted SVM approach shows even better classification effectiveness compared to the SVM method. For example, *ACC* and *TPR* increase to 93.2% and 86.5%. These comparisons demonstrate the superior performance of our proposed CFG guided Weighted SVM approach.

2) Case Study II — vim_codeinject: This case is also in the offline infection category, but the infection technique and the payload are different. We choose the hacking tool Codeinject [32] to inject a password dialog into a portable executable, in this case Vim is the host application. When the user starts Vim, a password dialog will be popped up asking for the password, which is pre-set when the trojaned binary is generated. If the user does not know the password, Vim exits silently.

From Figure 6, we can see that *vim_codeinject* increases in all five measurements for each classification model. For instance, *ACCs* for CGraph, SVM and WSVM are 35.5%, 72.5% and 85.2%. Another measurement, *NPVs* for CGraph, SVM and WSVM are 51.8%, 64.6% and 78.2%, respectively.

3) Case Study III — putty_reverse_https_online: This case is in the online injection category. In the event that there is some unpatched vulnerability in the target system, an adversary may craft some shellcode and perform a remote exploitation to run the shellcode. In order to stay persistent in the system, after taking control of the system, the adversaries can choose a long-running process and inject a backdoor payload into its memory space. They first allocate a memory slot for the backdoor payload and then remotely create a thread to run the code in parallel with the benign code. In this case, the adversaries first leverage the Metasploit Framework to take over the target system. Then they can run the script of post/windows/manage/payload_inject to inject the Meterpreter payload into the memory of a running Putty. Finally they can connect to the Meterpreter payload running within the Putty's process via a reverse HTTPS connection.

We can see from Figure 7 that the *ACC*, *PPV*, *TPR*, *TNR*, and *NPV* for WSVM are the highest, which is consistent with our observation in the Case Study I and II. For example, the corresponding *ACCs* for the three methods are 69.22%, 78.25% and 86.86%, and their respective *TPRs* are 41.2%, 56.1% and 73.8%.

VI. Discussion

In this section, we examine the limitations of LEAPS and propose potential solutions to address these problems. In addition, we discuss some future research opportunities in the area of attack detection by bridging program analysis and machine learning based techniques.

A. Source-level Trojaned Applications

LEAPS currently targets *camouflaged attacks* against binary applications, which indicates that the relative offsets of the benign code will not change. However, imagine that the adversary has obtained the source code of this benign application. He or she could add the source code of the malicious payload into the original code base, recompile the program, and deliver the trojaned application to the victim.

For closed-source software, only internal developers of the software vendors can intentionally conduct such trojan implanting attacks. For software in the open-source community, each line of the committed source code will be open to public inspection, which makes such attacks more difficult. Assuming there exist such malicious vendors or negligent maintainers, currently LEAPS is not able to assign correct weights in the *mixed dataset* because the CFG itself has been modified.

In order to address this limitation, we need to generalize our CFG comparison algorithm. For trojaned applications, assuming that the adversaries do not change the functionality of the original benign software (they just implant the payload's source code), the general structure of the benign subgraph in the CFG will not change. In light of this, instead of conducting exact matching, we could search for isomorphic subgraphs in both benign/mixed CFGs by identifying and aligning pivotal nodes. We consider this as our future work to improve LEAPS.

B. Future Work in Learning

LEAPS employs a Weighted SVM model to distinguish malicious events from benign ones. As shown in the experimental results, LEAPS achieves reasonably good performance on *camouflaged attack* detection, and consistently outperforms approaches based on system-level call graph and pure SVM. However, LEAPS only takes the order of adjacent events into account. But in real scenarios, there may exist some causal relations between multiple events dispersed far away (temporally) in the log. Therefore, we plan to explore more machine learning techniques, such as conditional random field model and hidden Markov model, to reveal such hidden relationships between events.

VII. Related Work

Host-based anomaly detection and malware classification systems are well-researched in recent years. The general procedure of these approaches is to extract the execution abstraction from a subject program, build a model, and use this model to make decisions on future data.

Some systems are based on the assumption that source code or binary is available for analysis, thus they are able to derive a precise model to represent the program's execution. Wagner et al. [1] define a model of expected application behavior through static analysis of its source code, and then check the

system call trace for compliance at runtime. Giffin et al. [2], [3] introduce the *Dyck* model, based on static binary analysis, to include program instrumentation on the binary to facilitate efficient runtime monitoring. DOME [4] first identifies the locations of system calls within the executables using static analysis, and then verify at runtime that each observed system call is invoked from its legitimate call site. SMIT [33] is a malware indexing system that leverages an executable's function-call graphs to cluster malware. Kruegel et al. [34] propose extracting CFGs from worm executables embedded in the network stream to identify structural similarities among polymorphic worms. In real-world scenarios, source code or executables may not always be available for training. Furthermore, obfuscated executables and complexity of binary disassembly render static analysis difficult to build accurate models. In comparison, LEAPS does not require static analysis or instrumentation of application source or binary code. We model the execution of the program only by analyzing the system event log and infer its CFG to guide statistical learning.

Some researchers also propose black-box or gray-box approaches to infer the execution model without static analysis. For example, Sekar et al. [6] propose an approach to generate a deterministic FSA by monitoring the normal program executions at runtime, thus avoiding static analysis on source code. Gao et al. [7] propose a gray-box approach that builds execution graphs based on system call sequences and does not require static analysis. Feng et al. [8] propose extracting return addresses from the call stack to build a model of abstract execution path and use the model to detect exploits. LEAPS shares the methodology of dynamically deriving the program execution model. Yet it is among the first efforts to leverage the inferred execution models to refine statistical learning models by pruning noisy training datasets.

Statistical learning techniques are also widely adopted in anomaly detection research. Such techniques have the advantage of being robust in processing incomplete training data, thus they can usually achieve better classification results. The input of these systems is based on the interaction between the applications and OS (e.g., system call sequence, system state change, and access activities). For example, Hofmeyr et al. [11] propose to characterize normal behaviors of a program in terms of system call sequences, thus they can detect an anomalous execution if it produces aberrant system call sequences. Wespi et al. [12] leverage *Teiresias*, an algorithm for discovering patterns in unaligned biological sequences, to build a table of variable-length patterns of audit events. Lee et al. [9], [10] leverage data mining techniques to find patterns of system features that describe program behavior. Bailey et al. [23] develop a classification technique that categorizes malware behavior in terms of system state changes, rather than from system call patterns. Lanzi et al. [35] demonstrate that malware detectors based on system call sequence may not be effective in real-world scenarios and build a model based on access activities on files and the registry.

Recently, some researchers introduce more sophisticated machine learning models, such as HMM and SVM, to assist classification. Warrender et al. [13] compare four anomaly detection models based on the system call dataset and conclude that HMM achieves the best accuracy on average, but with high computational costs. Gao et al. [36] propose the concept of *behavioral distance* to compare the differences of process' behaviors on different platforms based on system calls invoked. In subsequent work [14], they also introduce HMM to measure the *behavior distance* to better account for system call orderings. Heller et al. [16] use a one-class SVM to perform training on a dataset of normal registry accesses and then detect anomalous registry behavior in the testing data. Kolter et al. [37] use n-grams of byte codes from benign/malicious executables as features and evaluate them on a variety of inductive methods to train the classification model. Rieck et al. [15] extract behavior of malware in a sandbox environment and use SVM to learn the classification model for discriminating malware types. Bayer et al. [38] leverage *locality sensitive hashing* to perform unsupervised clustering based on the malware's behavior extracted in a controlled environment. Khan et al. [17] present a study on using hierarchical clustering analysis for enhancing the training time of SVM, especially for dealing with large data sets in intrusion detection. Eskin [39] also recognizes the existence of noisy training datasets. His solution is to first learn a distribution probability over training data and then apply a statistical test to detect anomalies. LEAPS also adopts SVM as the statistical learning model. However, different from these efforts that are purely based on learning, LEAPS leverages the inferred CFGs as guidance to prune the noisy datasets, thus effectively boosting the accuracy of the learned model for detecting *camouflaged attacks*.

VIII. Conclusion

Camouflaged attacks implant malicious payloads into benign applications and execute concurrently under the cover of benign processes. This causes traditional statistical learning based detection systems to generate a misleading decision boundary due to noisy training data. In this paper, we present LEAPS, a new attack detection system based on a supervised statistical learning model to classify benign and malicious system events. Different from existing approaches, LEAPS leverages CFGs inferred from system event logs as guidance to automatically refine noisy training data, leading to a more accurate classification model for *camouflaged attack* detection. We have conducted extensive evaluation on a range of real-world attacks with offline and online camouflaging strategy. Our experimental results demonstrate that LEAPS can effectively improve classification accuracy compared to traditional learning and system-level call graph based models.

Acknowledgment

This work was inspired by technical discussions with Dr. Sukarno Mertoguno, who proposed the "Learn-2-Reason" paradigm [19]. We also thank Brendan Saltaformaggio and the anonymous reviewers for their constructive comments. This research has been supported in part by ONR under Award N000141410468, NSF under Award 1409668, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*,

ser. SP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 156–.

[2] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection." in *NDSS*, 2004.

[3] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 61–79.

[4] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham, "Detection of injected, dynamically generated, and obfuscated malicious code," in *Proceedings of the 2003 ACM Workshop on Rapid Malcode*, ser. WORM '03. New York, NY, USA: ACM, 2003, pp. 76–82.

[5] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, May 2004, pp. 194–208.

[6] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, ser. SP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 144–.

[7] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 318–329.

[8] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, ser. SP '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 62–.

[9] W. Lee, S. J. Stolfo, and P. K. Chan, "Learning patterns from unix process execution traces for intrusion detection," in *In AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*. AAAI Press, 1997, pp. 50–56.

[10] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, ser. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, pp. 6–6.

[11] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, Aug. 1998.

[12] A. Wespi, M. Dacier, and H. Debar, "Intrusion detection using variable-length audit trail patterns," in *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, ser. RAID '00. London, UK, UK: Springer-Verlag, 2000, pp. 110–129.

[13] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: alternative data models," in *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, 1999, pp. 133–145.

[14] D. Gao, M. K. Reiter, and D. Song, "Behavioral distance measurement using hidden markov models," in *Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 19–40.

[15] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 108–125.

[16] K. A. Heller, K. M. Svore, A. D. Keromytis, and S. J. Stolfo, "One class support vector machines for detecting anomalous windows registry accesses," in *In Proc. of the workshop on Data Mining for Computer Security*, 2003.

[17] L. Khan, M. Awad, and B. Thuraisingham, "A new intrusion detection system using support vector machines and hierarchical clustering," *The VLDB Journal*, vol. 16, no. 4, pp. 507–521, Oct. 2007.

[18] S.-J. Horng, M.-Y. Su, Y.-H. Chen, T.-W. Kao, R.-J. Chen, J.-L. Lai, and C. D. Perkasa, "A novel intrusion detection system based on hierarchical clustering and support vector machines," *Expert systems with Applications*, vol. 38, no. 1, pp. 306–313, 2011.

[19] J. S. Mertoguno, "Human decision making model for autonomic cyber systems," *International Journal on Artificial Intelligence Tools Vol. 23, No. 6 (2014)*.

[20] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, and D. Xu, "Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces," in *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '14. New York, NY, USA: ACM, 2014, pp. 235–247.

[21] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. Springer New York Inc., 2001.

[22] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "Byteweight: Learning to recognize functions in binary code," pp. 845–860, 2014.

[23] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 178–197.

[24] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proceedings of the 18th Conference on USENIX Security Symposium*, ser. SSYM'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 351–366.

[25] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[26] C. Chang and C. Lin, "LIBSVM: A library for support vector machines," *ACM TIST*, vol. 2, no. 3, p. 27, 2011.

[27] S. hyuk Cha, "A genetic algorithm for constructing compact binary decision trees," *Journal of Pattern Recognition Research*, 2009.

[28] B. Schölkopf, R. Herbrich, and A. J. Smola, "A generalized representer theorem," in *COLT*, 2001, pp. 416–426.

[29] I. Buch and R. Park, "Improve debugging and performance tuning with etw," *MSDN Magazine,[Online],Avaliable from: http://msdn.microsoft.com/en-us/magazine/cc163437.aspx*, 2007.

[30] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote sensing of Environment*, vol. 62, no. 1, pp. 77–89, 1997.

[31] "Metasploit," http://www.metasploit.com/.

[32] "Portable Executable (P.E.) Code Injection: Injecting an Entire C Compiled Application," http://www.codeproject.com/Articles/24417/Portable-Executable-P-E-Code-Injection-Injecting-a.

[33] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 611–620.

[34] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 207–226.

[35] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "Accessminer: Using system-centric models for malware protection," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 399–412.

[36] D. Gao, M. K. Reiter, and D. Song, "Behavioral distance for intrusion detection," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 63–81.

[37] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, Dec. 2006.

[38] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[39] E. Eskin, "Anomaly detection over noisy data using learned probability distributions," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 255–262.