Ensemble: Community-Based Anomaly Detection for Popular Applications

Feng Qian, Zhiyun Qian, Z. Morley Mao, and Atul Prakash

University of Michigan, Ann Arbor MI 48109, USA
{fengqian,zhiyunq,zmao,aprakash}@umich.edu

Abstract. A major challenge in securing end-user systems is the risk of popular applications being hijacked at run-time. Traditional measures do not prevent such threats because the code itself is unmodified and local anomaly detectors are difficult to tune for correct thresholds due to insufficient training data.

Given that the target of attackers are often popular applications for communication and social networking, we propose *Ensemble*, a novel, automated approach based on a trusted community of users contributing system-call level local behavioral profiles of their applications to a global profile merging engine. The trust can be assumed in cases such as enterprise environments and can be further policed by reputation systems, *e.g.,* by exploiting trust relationships inherently associated with social networks. The generated global profile can be used by all community users for local anomaly detection or prevention. Evaluation results based on a malware pool of 57 exploits demonstrate that Ensemble is an effective defense technique for communities of about 300 or more users as in enterprise environments.

1 Introduction

End-user systems can be difficult to secure for a variety of reasons. They are typically unmanaged: users download software, browser bugs, *etc.* In this paper, we focus on defending against a class of attacks in which popular applications are hijacked at run-time. In the past, this has led to wide-spread attacks such as the Skype worm [14] spread using Skype and buffer overflows in Outlook email clients to execute arbitrary code [7]. Traditional measures, such as anti-virus scanners [5], do not prevent such threats because the application code itself is unmodified. Prior work indicates that system-call level profiling [23,33,37] may help detect such attacks early but a significant barrier is a lack of sufficient training data to ensure low false positive rates.

In this paper, we present *Ensemble*, a novel unsupervised anomaly detection approach based on the idea of a trusted community of users contributing system-call level *local profiles* of an application to a common merging engine. The merging engine generates a *global profile* that captures the possible space of normal run-time behaviors of an application. The global profile can be used to detect or prevent anomalies in application behavior at each end-host in real time. The promise of this approach is that it helps overcome the problem of a lack of sufficient training data at each host and can be largely automated. The challenges are making such a system efficient, overcoming the differences in profiles due to factors such as variations in installation directories or hardware, and identifying the appropriate information to collect in profiles.

© Institute for Computer Science, Social-Informatics and Telecommunications Engineering 2009

The underlying hypothesis of Ensemble is that, *as the number of local profiles increases, the aggregate global profile tends to converge, thus revealing the normal behavior of the target application.* Most applications in our experiments were found to satisfy this property, though we also identified types of applications that would be exceptions. This paper makes the following contributions.

Handling diversity in execution environments. Various factors impact community-based profiling, *e.g.,* the same application at different hosts may be installed in different directories, run with different amount of memory, and use different number of CPUs. All these can cause variations in the system call traces with their parameters. We determined the types of data to use for generating behavioral profiles to handle these variations, while keeping profiles compact and representative of the application.

Analysis of the relationship between the community size and false positive rates. We first applied community-based anomaly detection to a community of 12 users using a normal, clean instant messaging application. The detailed system-call level data were sampled for 50 minutes during 5 hours with each local profile generated based on one minute of sampled data. We found that high false positive rates to be of significant concern, just as with single-host profiling using system calls. A testbed of virtual machines was subsequently used to study the impact of scaling up the system to a larger user community. We found that the techniques, in general, tend to become much more effective with larger community size. Significant reduction in false positive rates was observed after reaching approximately 300 users.

Techniques to reduce data transfer by sharing summary data generated by profiling applications. We show that while each host collects detailed system-call level data [23,26,36] for local analysis, it only needs to send a modest amount of local profile data per application (approximately, 4-5 KB/sec) to a common server to create community profiles.

A general interface. Our system provides a useful abstraction of a general interface for any target application to be protected. Multiple applications can subscribe to the Ensemble service.

Ensemble is currently implemented in user space in Windows. We used Detour library [27] by Microsoft Research to intercept system calls for target applications. For improved efficiency, as discussed in §4.2, Ensemble can be implemented as a service in the OS kernel. The rest of the paper is organized as follows: §2 overviews the related work; §3 describes the overall model of Ensemble; §4 details our implementation; and §5 evaluates the system experimentally. Finally, §6 discusses limitations before concluding in §7.

2 Related Work

Our work improves on existing work in the area of anomaly detection by exploring the applicability of community-based profiling to generate detailed run-time behavior profiles of applications at the system call level. Below we highlight some of the related approaches in malware detection and containment.

Anomaly Detection. One of the first studies on anomaly detection for applications was done by Forrest *et al.* [23,26,36]. They executed an application multiple times with different inputs to collect system call sequences and then used those to form the baseline behavior of the program. Any significant deviation from the baseline was considered as an anomaly. Many of the follow-up studies [16,24,21,25,37,33,20] incorporate machine learning techniques such as hidden-Markov model and neural networks. Later studies examined the inclusion of system call arguments [13] and call stack information [22]. Generating a common model from different runs is a non-trivial problem. In [16], Ballardie and Crowcroft explore several representative models, including frequency-based models, a data-mining approach, and a finite state machine approach.

All these above approaches can suffer from high false positive rate. The data collection process is typically manual or may take a long time to cover most normal behavior. If the application's normal behaviors are not adequately captured, unobserved normal behavior is likely misclassified as abnormal. While better machine learning algorithms [25,33] can help, one fundamental problem in making these schemes practical is the difficulty in getting sufficient training data to capture comprehensive application behavior.

Our work builds on the approaches in the above systems. The primary contribution is to show that if a large user community sharing their training data with an IDS at a fine-grained level, behavioral profiles can be generated that are much more complete and accurate than local profiles. One of the challenges we examined in extending the techniques to a community environment is that not just the inputs, but the operating environment for the software can be different. In our experiments, we allowed applications to be installed in random directories on various systems with diverse hardware configuration and varying workload imposed by other applications. We extend existing algorithms for combining profiles to handle likely variations.

Community-based Systems. The concept of "application community" [2] has been proposed to collaboratively diagnose and respond to attacks by generating appropriate configuration patches and filters. The goal is to generate a community-specific situation awareness gauge to predict imminent attacks. But it does not focus on anomaly detection as in our work to help prevent attacks.

A similar concept of "collaborative learning for security" [19] is applied to automatically generate a patch to the problematic software without affecting application functionality. However, the detectors used are static detectors without training, and the ways in which the community is utilized are limited to gathering detailed execution constraints in the binary, distributing the generated patch, and letting the user community evaluate them.

Companies, such as Symantec [12], Microsoft, and Google also leverage the notion of a community to help identify malware programs or spam emails [4] from user based feedback. Vigilante [17] and Sweeper [34] try to contain Internet worms by automatically detecting exploits. Both enable a user community to share their antibodies to prevent and stop future attacks from Internet worms.

In other application contexts, the concept of community has also been explored. PeerPressure [35] utilizes it to automatically detect and troubleshoot misconfigurations by assuming that most users in the community have the correct configuration. The Gamma System [32] was proposed to split the monitoring task among community users, enabling minimally intrusive program analysis and software evolution. Similarly, Cooperative Bug Isolation [31,30] leverages the community to do "statistical debugging" based on the feedback data automatically generated by community users.

In contrast to the above body of work, our work examines the effectiveness of applying the notion of community at a much finer-grained level. Instead of just combining binary feedback or signatures of worms, we integrate run-time behavioral profiles, consisting of system calls and associated parameters, of applications across a community of heterogeneous users. This allows us to extend anomaly detection to additional classes of software applications.

Signature based anti-virus (AV) software. In this approach, a user typically uses a signature database of known attacks, resulting in the advantage of negligible false positives. Unfortunately, it is difficult to maintain signatures covering new attacks. A study by Oberheide *et al.* [28] found that commercial AV software has a detection rate ranging from only 54.9% to 86.6% for attacks that occurred in the previous year. More importantly, the AV software had significantly poorer detection rates for more recent malware samples. This implies that anomaly based detection is still indispensable.

Behavior-based intrusion detection systems (IDS). These systems rely on pre-defined rules to detect anomalies in the run-time system behavior. They can better detect zero-day attacks that attempt to evade code-based signatures. But, getting the rules right can be difficult and therefore the rules tend to be relatively coarse-grained. For example, by default, McAfee VirusScan Enterprise 8.5i [5] Access Protection rule blocks outbound port 25 to filter malicious email programs. However, to get normal email applications to work, 42 popular email clients, such as `outlook.exe` and `thunderbird.exe` [11], are exempt. Note these applications are often the ones exploited.

3 Methodology

In this section, first we present high-level methodologies used in *Ensemble*, then explain them in detail in §3.1 to §3.3.

The goal of Ensemble is to detect application misbehavior, particularly caused by zero-day attacks. As the start point of our approach, we generate a *local profile* for each application instance. A *profile* is a summary of target application's inter-process communications and its behavior that can result in persistent changes (changes that survive across reboots) to the file system, the registry, network, and other system settings. They are abstracted from system call traces. Statistically, it can be seen as representative data points in the sample space containing all possible state changing behavior of the target application.

We envision that a large number of community users feed local profiles of an application to a central server, which periodically aggregates them into a *global profile*, depicting the application's normal behavior as a baseline. The global profile serves as a *classifier* that identifies anomalies using collected local profiles as training data.

To detect and prevent intrusion, we monitor the application behavior and compared it with the global profile continuously. An alarm is triggered when the application is about to perform an operation that does not match the global profile. The user can be alerted or the system can be configured to directly block the operation. Next we investigate several important challenges of our methodology.

3.1 Profile Generation

Local profiles. A local profile is generated from raw system call traces [26]. In Windows, system calls are undocumented, thus we use Windows API calls in our prototype. For simplicity we ignore a set of APIs that do not modify host file system or network state such as graphics and user interface API that are unlikely abused or even if abused will likely be visible through other APIs we monitor. Also, we only focus on operations executed by the target application given the profile is for a particular application, with the exception of the process dependency, as discussed below.

Global profiles. A global profile is distilled from multiple local profiles. We develop a taxonomy for APIs in terms of functionality (process dependency, file access, network access, *etc.*). For each category, corresponding records in local profiles are aggregated by key attributes (Table 1). An example of aggregating File Access category is shown in Table 2.

Table 1. Key attributes for primary categories in global profiles

Category	Key Attributes
Process Dependency	Src Process Name/Image Hash, Dst Process Name/Image Hash, Type $\in$ {Fork, Hook, File...}
File Access	Filename, Type $\in$ {Read, Write}
Registry Access	Registry key, Type $\in$ {Read, Write}
Network Connection	Remote IP, Remote Port, Protocol $\in$ {TCP, UDP, other}

Table 2. Example: aggregate records in local profile (a) into global profile (b)

(a) Local profiles

Profile ID	Filename	Bytes accessed	Type
1	a.dat	10	read
1	a.dat	15	read
1	b.dat	10	read
2	b.dat	10	read

(b) Global profiles

Filename	Type	Count by profiles
a.dat	read	1
b.dat	read	2

Among all the categories, the process dependency [29] depicts the interaction among processes of the target application and other processes. A local profile contains two types of dependencies: indirect and direct dependency. Indirect dependency, such as a file dependency (Process A writes file F, which is then read by Process B), requires an object (*e.g.,* a file or an IP address) as an intermediary. It is synthesized by correlating multiple API calls. Direct dependency, such as a fork dependency, takes place without an intermediary. It can be inferred from a single API call.

3.2 The Environment Diversity Challenge

For categories other than process dependency, the simplified methodology illustrated in Table 2 has limitations. For example, for a text processor, different users edit different files, thus the file access category is not aggregatable if naively using the filename as the key attribute. Similarly, a P2P client may talk to random IP addresses, leading the aggregation in the global profile to be a set of IP addresses each with very few occurrences. We apply two methods to address this challenge.

First, we use predefined rules to normalize the path and file names. For example, `c:\Documents and Settings\Alice\a.dat` is normalized to *`USER-DOC`*`\a.dat`. This also helps protect the privacy of community users.

Second, our main solution is *Stack Signature*, which describes the stack history of the calling thread for each API call. The key idea is that the "random" events of the same functionality of a program such as sending a message or making a VoIP call in Skype, should be associated with a fixed set of execution paths that can be represented by call stacks. Based on this assumption, we introduce Stack Signature, a compact version of call stack. A Stack Signature is calculated by iterating all stack frames of the current thread and XORing their return addresses. In the case of recursive calls, return addresses occurring multiple times are counted once.

In a global profile, the relationship between stack signatures and objects (*e.g.,* filenames and IP addresses) can be characterized by a weighted bipartite graph, whose vertices are divided into two disjoint sets X and Y, where X is the set of stack signatures and Y is the set of objects. There is an edge $e : x \to y \in E$ where $x \in X$ and $y \in Y$, if and only if an event accessing object y has stack signature x in at least one local profile. Each element in X, Y and E has a weight, indicating its occurrence frequency in terms of the number of local profiles. Except for the process dependency which is fairly stable, we introduce stack signatures and use bipartite graphs as the data abstraction for all other categories.

We observe many such cases in our experiments. For example, at stack signature `0x61AE46F8`, QQ [8] – an instant messaging application may receive data from at least 64 different servers such as 121.14.*.*, 219.133.*.*, 58.61.*.*, via port 8000. All servers are found at Guangdong, China, where the headquarter of QQ is located. The size of received data is always a multiple of 10240 bytes.

3.3 Anomaly Detection

As described at the beginning of this section, Ensemble clients periodically pull the global profile from the server. The anomaly detection and prevention are performed continuously. Before each operation monitored by Ensemble is executed, the API call is intercepted and compared with the global profile using the following comparison algorithm.

1. **Threshold-based process dependency anomaly detection.** If a process dependency D is detected (*e.g.,* a *fork* or *file* dependency), we locate its frequency $f(D) = \frac{\text{\# of local profiles containing } D}{\text{\# of local profiles}}$ in the global profile, if $f(D) < th_{PD}$, where th_{PD} is a threshold, then D is regarded as abnormal.

2. **Stack signature analysis.** If the operation to be executed by the target application falls into other categories in Table 1, then its stack signature x is calculated, its object y is identified, and $e : x \rightarrow y$ is matched against the bipartite graph $B_G = \{X_G, Y_G\}$ in the global profile. Let the frequency of e and x in B_G be $f(e)$ and $f(x)$, respectively. (*i.e.,* $f(e) = \frac{\text{\# of local profiles containing } e}{\text{\# of local profiles}}$). Let the degree of x in B_G be $d(x)$. We also introduce thresholds th_e, th_x and deg_x. We determine whether e is an abnormal action by several tests searching for the predictable relation of the objects accessed by stack signatures.

Test 1. Does a fixed stack signature always access a fixed object? (*e.g.,* The program reads a constant configuration file) Formally, if $f(e) > th_e$, then e passes the test and no further tests are needed.

Test 2. Does a fixed stack signature always access different objects? (*e.g.,* A file editor may open different files) Formally, if $f(x) > th_x$ and $d(x) > deg_x$, then e passes the test and no further tests are needed. This handles the "the Environment Diversity Challenge."

Some challenges arise, as we observe that in multiple executions of the same application, a single object may be accessed by different stack signatures forming one or more clusters. Figure 1 is an example of reading file `ServUCert.key` in 1,305 executions by Serv-U 5.0.0.0 (a commercial FTP server). The stack signatures form a cluster ranging from `0x1019A500` to `0x1019A5FF`. We conjecture two reasons: (1) The locality of object access. The same object is often accessed at close-by instruction addresses. For example, the code in Figure 2 is common in C programs. The consecutive calls of `fread` satisfy the locality principle. (2) The accumulation of varieties. A

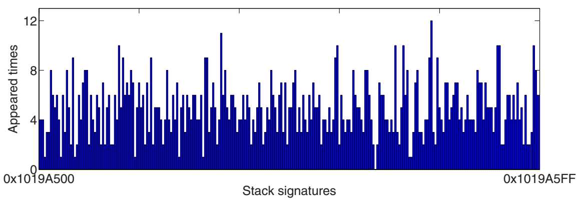
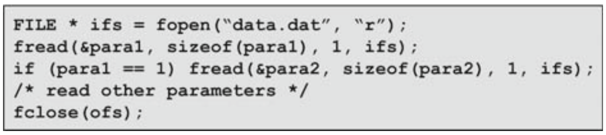


Fig. 1. Frequency of accessing `ServUCert.key` from different stack signatures in 1305 local profiles



```
FILE * ifs = fopen("data.dat", "r");
fread(&para1, sizeof(para1), 1, ifs);
if (para1 == 1) fread(&para2, sizeof(para2), 1, ifs);
/* read other parameters */
fclose(ofs);
```

Fig. 2. Sample code of reading a file

signature is calculated by XORing return addresses of n stack frames with each frame having a variety of k_i, the total variety can be as large as $\prod_{i=1}^{n} k_i$.

Motivated by the above observation, we add two additional tests to reduce false positives.

Test 3. Does a cluster of stack signatures access a fixed object? We define a cluster by a window centering at x: $X_{win} = \{z \in X_G \,||z - x| \leq winSize\}$. Formally, if $\sum_{z \in X_{win}} f(e' : z \rightarrow y) > th_e$, then $e : x \rightarrow y$ passes this test.

Test 4. Does a cluster of stack signatures access different objects? Formally, if $\sum_{z \in X_{win}} f(z) > th_x$ and $\sum_{z \in X_{win}} d(z) > deg_x$, then e passes this test. It is a further generalization of Test 3.

Test 3 and 4 may introduce false negatives; however, they are expedient alternatives in the situation where the number of samples is limited. Ideally, when the global profile contains a large enough sample space, Test 3 and 4 can be replaced by Test 1 and 2, respectively, since the range of stack signatures is finite. Figure 3 illustrates four patterns in the global profile, corresponding to the above four tests.

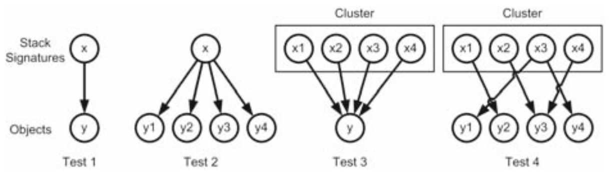


Fig. 3. Four API invocation patterns

4 Implementation

The architecture of our Ensemble prototype is illustrated in Figure 4. It is designed to perform online anomaly detection using continuously updated global profiles and generated local profiles. Existing work is mostly evaluated in Linux environments while our system is implemented on Microsoft Windows XP, which is a more common attack target. Our prototype is implemented using about 10,000 lines of C++ code.

In our design, we initially tried to implement Ensemble by using system call sequences (N-gram previously proposed [23,26,36]) as the representation of local profiles, due to its claimed effectiveness and simplicity. However, we found that N-gram has surprisingly low convergence speed for Windows API sequences in terms of obtaining the model of application's normal behaviors, likely due to a much larger sample space than in Linux (the number of Windows APIs is 6 times the number of Linux syscalls). We estimate two reasons for such big discrepancy: first, there are distinct difference between Unix/Linux system calls and Windows APIs; second, modern applications are becoming more and more complicated. System calls may be a too find-grained characterization of program behavior. Note that a lot of researchers apply N-gram algorithm on virus or malwares, whose binary sizes are much less than legitimate applications. Therefore, instead we resort to the simpler frequency-based model as described in §3.1 that has a faster convergence behavior.

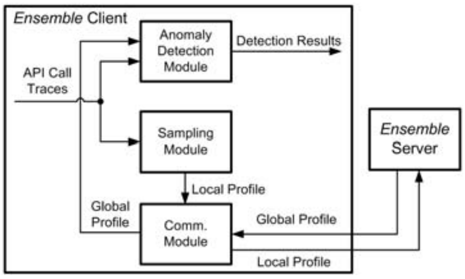


Fig. 4. The Ensemble Architecture

4.1 Generating Profiles and Anomaly Detection

We used the *Detour* Library [27] to monitor and log 106 APIs calls related to file system (26), registry (8), file mapping (6), messages (8), thread (4), process (8), network (13), pipe (6), hook (3), clipboard (3), system time (6), DNS (2), handle management (2) and user accounts management (11), most of which are Windows specific. To the best of our knowledge, they cover most APIs that can cause inter-process communications, or result in persistent changes to the file system, the registry, the network, and other system settings. Note that it is fairly easy to include new APIs to the framework. We generate stack signatures using the `StackWalk64` function in Windows Debugging Library.

Given the raw API traces and their stack signatures, the local profiles are generated as described in §3.1 (for process dependency) and §3.2 (for other categories). We implemented seven categories for profiles. (1) process dependency, (2) file access, (3) directory access, (4) registry access, (5) network connection, (6) DNS, and (7) IP prefix access. For (1), we handle 4 types of direct process dependencies: send message, set hook, create/terminate/suspend process (thread) and write/read/alloc/dealloc process memory, and 8 types of indirect dependencies: files, registry, file mapping, network, named pipes, anonymous pipes, system time and clipboard. The transformation from API traces to other categories (*e.g.,* file access, network access) is trivially done by translating API parameters.

The global profile is generated by grouping various local profiles. Except for the process dependency, which is represented by a table like Table 2(b), other categories are represented using bipartite graphs (stack signature $\rightarrow$ object names).

Our anomaly detection algorithm described in §3.3 is very efficient. For process dependency, the dependency inference and frequency look up is $O(1)$ in run time using hash tables. For other categories using bipartite graphs, the computational complexity for Tests 1 and 2 is $O(1)$; while Test 3 and 4 are also $O(1)$ given that the window size is a small constant.

4.2 Operational Model

Finally, we present an overview of Ensemble's operational model. At each client, Ensemble is running as a system service and is transparent to the target application. CAPTCHA is used when subscribing or unsubscribing Ensemble services to prevent tampering from bots.

When the application is running, the *Ensemble sampling module* periodically logs its API calls with stack signatures[1] and generates the local profile (*e.g.,* every 3 hours, one local profile is generated from 1-min sampling of API call traces). The *Ensemble communication module* periodically submits the local profile to the server, and also fetches the global profile from it. The *Ensemble Anomaly Detection Module* keeps monitoring target application's API calls and matching them with the global profile. If an alarm is triggered, the requested operation is denied, or the decision is left to the user.

Initially our anomaly detection is sampled: a local profile is generated periodically and compared with the global profile. Then we found that even if the anomaly detection is performed continuously, the extra overhead is acceptable (less than 2%), given that in most cases, the applications' API calls are not invoked in a "bursty" manner.

The Ensemble server can be maintained either on a large scale (*e.g.,* by the application vendor), or on a small scale (*e.g.,* within an enterprise network). Its tasks include collecting local profiles, generating the global profile and other management functionalities. Ideally, each version of the application should have its own global profile. Depending on the specific application, one global profile may also characterize several versions with minor differences.

4.3 Limitations of the Prototype

Our current prototype has the following limitations which are not fundamental to our design. At the client side, the sampling module is implemented at the user level, using a third-party library. For future work we plan to move the entire system into Windows kernel. At the server side, in order to prevent pollution of global profiles, we plan to investigate the use of reputation systems that establish trust among community users. Currently, we envision our system to be mainly deployed in enterprise environments where trust can be assumed.

The latest Windows Vista adopts Address Space Load Randomization (ASLR) technique [1], which hampers the functionality of Stack Signatures. We can address this problem by using the relative offset of the return address from the module's start address, together with the module signature. We plan to explore this as future work.

5 Evaluation and Experiments

In this section, we systematically evaluate Ensemble. First we describe a small-scale deployment for a community of 12 users (§5.1). Based on the negative results due to the limited size of the community, we introduce our testbed and target applications used for experiments (§5.2), then analyze the generated local profiles (§5.3) and the resulting global profiles (§5.4). Next, we measure false positives (§5.5) and estimate false negatives using a recent malware collection (§5.6). Finally we present the performance evaluation of our system (§5.7).

5.1 Small Scale Real Deployment

We deployed Ensemble among 12 real users, using *Windows Live Messenger* (MSN) as the target application. All users were using Win XP SP2 but with different software

[1] To capture process dependency, some APIs called by other processes also need to be logged.

and hardware configurations. Before the experiment, we manually upgraded their MSN to the same version (2008 Build 8.5.1302.1018) and ensured the systems are virus-free. Users were not familiar with technical details of Ensemble, and were told to use MSN as usual. For each user, we collected 50 API call traces, each lasting 1 minute, during a 5-hour period. We used this dataset to evaluate false positives.

We used 5-fold cross validation on 600 traces to evaluate false positives. For each trace in the test group, if any API call triggered a false alarm, then the local profile was counted as one false positive. For the parameters in §3.3, we empirically set $th_e = 1\%, th_x = 1\%, deg_x = 10, winSize = 4KB$ (We tried different parameters such that $th_e < 2\%, th_x < 2\%, deg_x < 20$, and obtained similar results). We found that the false positive rates were too high to be accepted (greater than 30% for file access and registry access). The reason is that 12 users are not sufficient to form a community to cover diverse application behavior.

5.2 Experimental Infrastructure

To test the impact of a larger community, we created an automated testbed to simulate a community environment. The idea is simple: to execute the target application multiple times on the testbed. In each execution, a local profile is created and fed to the global profile generator, as if it was submitted by a real community user. Then we use the global profile to test against normal and abnormal behaviors and evaluate false positives and negatives. We have two design goals for the testbed.

- **Diverse User Behaviors.** Random user actions are injected during each trial. The distribution of the randomness should roughly conform to that of a real community.
- **Diverse System Environment.** During each trial, the system environment should also vary to simulate hardware and software variations in a real community. For example, a VoIP client may adjust its voice encoding strategy according to available network bandwidth, leading to different local profiles.

We manually created a Finite State Machine (FSM) for each target application to describe most of its main functionalities from an end user's perspective. FSM can be generated in a more automated fashion by combining user traces and adding some perturbation to include additional usage behavior. Despite the manual effort, FSM based representation for understanding application usage, even approximate, can aid in generating more diverse usage scenarios for a given application. Figure 5 is a simplified FSM for MSN. In each automated execution, the testbed partially iterates the FSM based on a Markov chain model, which characterizes the popularity of application's different functionalities. Each state transition $S_x \rightarrow S_y$ in the FSM represents a user action. A weight is assigned to e indicating the probability that the next state is S_y given the current state is S_x. For example, in Figure 5, "Login" is the initial state where the user starts the application. The probability that the user successfully logs in ($\frac{10}{1+2+10} = 77\%$) is much higher than the probability that the user enters an invalid ID or password (8%).

The testbed not only randomly chooses the action, but also executes some actions with randomness. For instance, it is able to operate an instant messenger by selecting a random user and chatting with him/her via random text messages, emotion icons, handwritings or Flash winks. In another example, the "make phone call" action in Skype is carried out by dialing a number from 3000 toll-free numbers we collected.

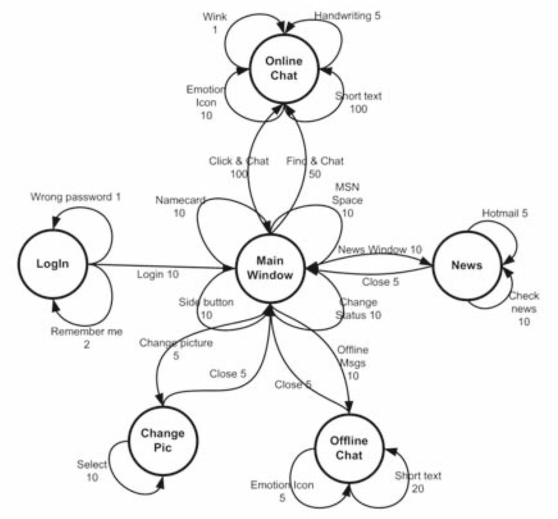


Fig. 5. A simplified finite state machine of MSN. Labels on edges indicate state transition probability.

We admit that our approach contains subjective elements and thus may not perfectly simulate a community environment. However, a community itself is a set of subjective users and has a tendency to change from time to time. Also, we will show in §5.3 the heavy-tailed distribution of simulated users' behaviors, which are usually the case in a real community.

To tackle the system environment randomness, the testbed automatically changes the hardware/software configurations for each trial. All experiments were conducted on virtual machines (VMware 6.0.2) for ease of management. The varied configuration includes memory, number of processors, installed software, existing running processes, system workload, firewall settings, system time, network bandwidth, DNS server, *etc.*

The testbed includes a FSM script parser, an action executor that maintains the state synchronization and sends mouse/keyboard input to the target application, a configuration manipulator that changes the system environment and a communicator that communicates with the Ensemble kernel. The testbed is built using about 3,000 lines of C++ code.

We chose four applications running on Microsoft Windows XP SP2 as our initial target applications: *Skype* 3.5.0.239; *Windows Live Messenger* (MSN) 2008 Build 8.5.1302.1018; *Tecnet QQ* [8] (2007 Beta 4, 7.0.374.204), an ICQ client with typically more than 30 million daily online users in China; *Serv-U* [9] (5.0.0.0), a commercial FTP server. These applications were selected due to their popularity and past history of attacks targeting them.

5.3 Local Profiles

Table 3 shows the number of local profiles, sampling times and API log sizes of local profiles of each target application. The sampling time was set to conform to a Gaussian

Table 3. Statistics of local profiles

Target App	# of local profiles	Sample Time (Mean)	Sample Time (Std Dev)	API Trace Size (Mean)	LP Size (Mean)
Skype	550	60 secs	5 secs	3.40MB	0.20MB
MSN	1298	75 secs	5 secs	1.17MB	0.09MB
QQ	1118	60 secs	5 secs	1.18MB	0.09MB
Serv-U	1305	45 secs	5 secs	0.23MB	0.03MB

Table 4. Statistics of global profiles

Target App	Process Dependency	File Read	File Write	Dir Read	Dir Write	Reg Read	Reg Write	Connections	IP Prefixes	DNS Query
Skype	8	209	237	178	208	4,587	328	135,844	115,864	0
MSN	10	2,884	244	795	90	54,506	2,749	6,417	554	0
QQ	4	6,549	8,029	6,541	8,021	59,491	229	11,867	9823	10,691
Serv-U	1	2,609	835	305	7	146	0	23,295	2	1

distribution. The sampling process started either at or after the application starts, and stopped either at or before the application terminates. The entire collection of local profiles lasted for one week.

As mentioned, we created randomness during each trial to simulate different user behavior in the community. Thus each "user" may explore a different subset of the application functionalities. Figure 6 illustrates the distribution of FSM patterns for Skype, MSN and QQ. A pattern defines the states iterated by the testbed in a single trial. If there are n possible states in FSM, then there exists $2^n - 1$ possible patterns $(0, 0, ..., 0, 1), ..., (1, 1, ..., 1, 1)$. For pattern $(a_1, a_2, ..., a_n)$, $a_i = 1$ iff the i-th state is visited at least once in a trial. The heavy-tailed distributions in Figure 6 demonstrate the diversity of user behaviors generated by our testbed, as well as the similarity of most users' behaviors. Although this may not exactly match the actual user behavior, we believe our method adds sufficient randomness to closely approximate general user activities.

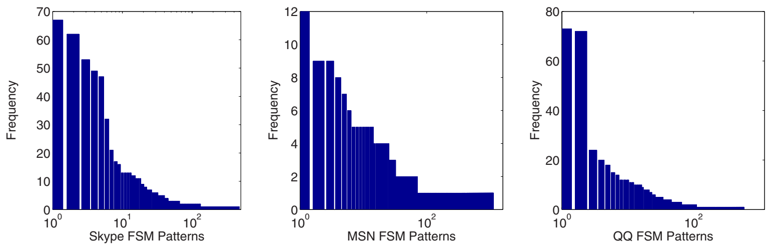


Fig. 6. FSM Pattern distribution for Skype (474 patterns), MSN (1137 patterns) and QQ (584 patterns). The X-axis is log-scaled.

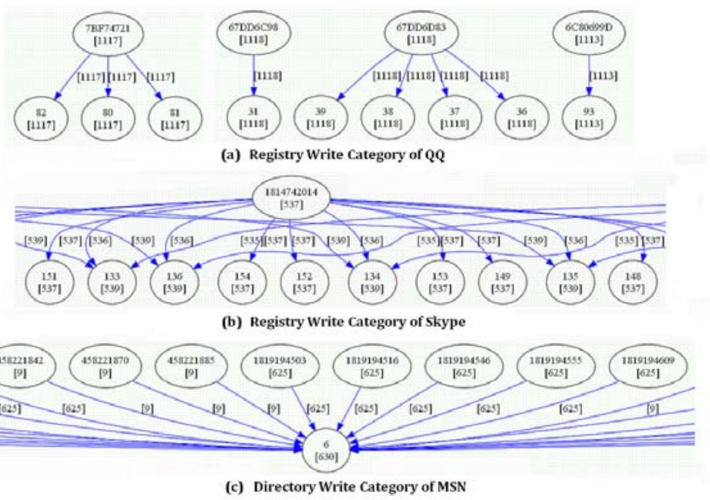
5.4 Global Profiles

Table 4 presents statistics of global profiles. The numbers in the table are the numbers of process dependencies and, for other categories, the number of edges in the bipartite graphs.

The process dependency categories of QQ, MSN and Skype are shown in Figures 9(a), 10, and 11(a), respectively. Only parts with solid line represent the observed dependencies; while the dotted lines indicate detected misbehavior (§5.6). The percentage on the edge denotes its occurrence frequency. The size of bipartite graphs is usually much larger.

Figure 7 shows examples of the bipartite graphs. For each subfigure, the upper part X is the set of stack signatures; the lower part Y is the set of objects (registry keys, directory names, *etc.*), which are represented by a number (object ID). The numbers in square brackets are the frequencies.

- Subfigure (a) is a common case where a fixed stack signature accesses a fixed object. For example, stack signature `0x7BF74721` always reads 3 registry keys:
  `\REGISTRY\MACHINE\SOFTWARE\Classes\QQCPHelper...`
  `\REGISTRY\MACHINE\SOFTWARE\Classes\CLSID\23752AA7...`
  `\REGISTRY\MACHINE\SOFTWARE\Classes\CLSID\23752AA7...`
- Subfigure (b) illustrates a random event problem. For each trial, Stack signature `1814742014 (0x6C2AC3FE)` writes different registry keys under `\REGISTRY\MACHINE\SOFTWARE\Classes\CLSID\` and `\REGISTRY\MACHINE\SOFTWARE\Classes\TypeLib\`.
- Subfigure (c) illustrates the slight variation of stack signatures, as explained in §3.3. We can observe two clusters of stack signatures in subfigure (c): 4582218??, 1819194???. Both clusters access the user cookie directory *USER-DOC*`\cookies`.



(a) Registry Write Category of QQ
(b) Registry Write Category of Skype
(c) Directory Write Category of MSN

Fig. 7. Examples of bipartite graphs. From top to bottom: (a) Registry write category of QQ (b) Registry write category of Skype (c) Directory write category of MSN.

5.5 False Positives

We used the same methodology (5-fold cross-validation) and the parameters as in the real deployment (§5.1) to evaluate the false positives for the testbed. In Table 5, the column "LPs" indicates the number of local profiles in the test group; the columns "Worst" and "Best" indicate the highest and lowest number of false positives (traces that contain at least one API call that triggers the false alarm), respectively, in 10 independent experiments (each experiment has 5 passes).

Table 5. Coarse-grained false positives (counting the number of local profiles)

Target App	Skype			MSN			QQ			ServU		
Category	LPs	Worst	Best	LPs	Worst	Best	LPs	Worst	Best	LPs	Worst	Best
Process Dependency	110	0	0	262	0	0	226	1	0	196	0	0
File Read	110	0	0	262	0	0	226	0	0	261	0	0
File Write	110	0	0	262	0	0	226	0	0	261	0	0
Directory Read	110	0	0	262	0	0	226	0	0	261	0	0
Directory Write	110	0	0	262	0	0	226	0	0	261	0	0
Registry Read	110	0	0	262	4	2	226	1	0	261	0	0
Registry Write	110	0	0	262	1	0	226	0	0	0	0	0
Connections		N/A		262	4	2	226	1	0	261	0	0
IP Prefixes		N/A		262	0	0	226	0	0	261	0	0
DNS Query	0	0	0	0	0	0	226	0	0	261	0	0

Table 6 presents a fine-grained false positive measurement. Similar as above, we employed 5-fold cross-validation and the experiment was repeated for 10 times using the same parameters. In Table 6, the column "Avg E" denotes the average number of API calls[2] in the test group, which were fed into Ensemble Anomaly Detection Module; the columns "Worst" and "Best" indicate the highest and lowest numbers of API calls that are mistakenly detected as abnormal, respectively.

For Skype and ServU, no false positives were observed. For MSN and QQ, although their fine-grained false positives of Registry Read and Connections categories were slightly higher even when the false positive rate converges (shown in Figure 8), the mistakenly detected API calls concentrated in a few local profiles (Upon manual inspection of the logs, it was highly possible that during the generation of these local profiles, the application terminated unexpectedly.). Ideally, if they were indeed application's natural behaviors, then as the pool of training data becomes larger, the initial "strange" behaviors will become normal, and the large size of training data is exactly the advantage of a community.

When we were testing Skype, it produced unacceptable false positive rates for network-related behavior (two categories whose false positives labeled as "N/A" in Table 5 and Table 6). Upon manual inspection, we found that the stack signatures from network related APIs were almost uniformly distributed in the entire address space, and

[2] To be precise, "Avg E" is the number of process dependencies or the number of edges in the bipartite graph.

Table 6. Fine-grained false positives. (counting the number of edges in PDGs or bipartite graphs)

Target App	Skype			MSN			QQ			ServU		
Category	Avg E	Worst	Best	Avg E	Worst	Best	Avg E	Worst	Best	Avg E	Worst	Best
Proc. Dep.	498	0	0	2203	0	0	844	1	0	196	0	0
File Read	13271	0	0	31650	0	0	40578	0	0	6290	0	0
File Write	1938	0	0	3623	0	0	40138	0	0	3473	0	0
Dir Read	10214	0	0	22292	0	0	39903	0	0	2758	0	0
Dir Write	1650	0	0	2711	0	0	40114	0	0	1810	0	0
Reg Read	43398	0	0	611294	55	37	415532	1	0	23943	0	0
Reg Write	33639	0	0	25441	1	0	23805	0	0	0	0	0
Connections		N/A		23398	12	4	18074	11	0	7194	0	0
IP Prefixes		N/A		17974	0	0	16385	0	0	516	0	0
DNS Query	0	0	0	0	0	0	17085	0	0	258	0	0

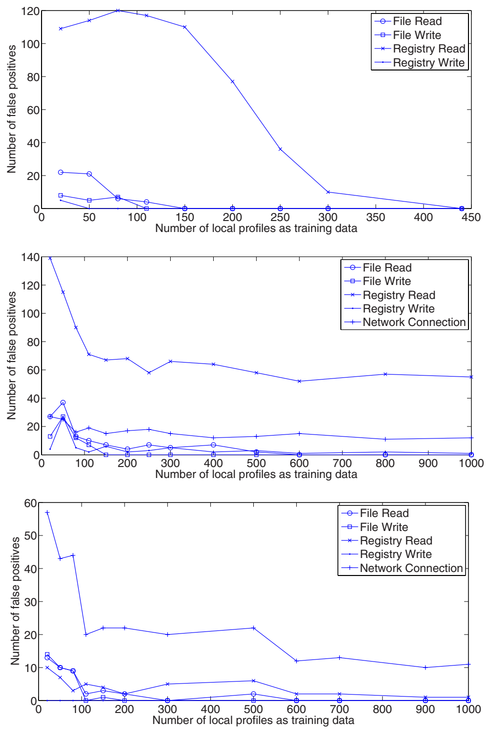


Fig. 8. Convergence of fine-grained FP as local profiles increase. (Top: Skype; Middle: MSN; Bottom: QQ)

the dumped stack frames were also abnormal. Based on our estimation, Skype may employ some obfuscation techniques to protect their code against reverse engineering [10]. In summary, we believe that the false positives of Ensemble are acceptable.

Furthermore, we used 600 API call traces obtained in real deployment to test against the global profile generated by 1,298 MSN local profiles from the testbed. We obtained false positive rates of 0% (process dependency), 6% (file read), 4% (file write), 2% (directory read), 1% (directory write), 11% (registry read), 6% (registry write), 9% (connections) and 3% (IP prefixes), using the metric in Table 5. Upon manual inspection, the main cause of false positives was the incompleteness of our FSM model, in which some use cases such as video chat were not covered.

We also measured the relationship between the community size and the false positive rate using a 5-fold cross-validation, and presents the results using the worst case (the highest number of false positives in 10 independent experiments). As shown in Figure 8 for three applications, it is clear that the fine-grained false positive rate significantly decreases with increasing number of local profiles, and converges to a stable value (We discussed the high false positives of QQ and MSN earlier in this section). A real active community is believed to have orders of magnitude of more local profiles submitted by users, thus ensuring a low false positive rate.

5.6 False Negatives

We evaluate false negatives on a total of 57 known malware programs and exploits for each target application by performing online comparison between the application behavior monitored in real time and the global profile, which was generated from local profiles described in Table 3. We used the same parameters as in the false positive evaluation.

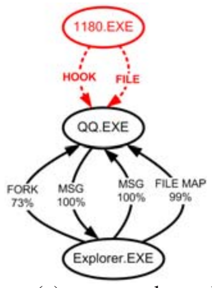
Table 7 summarizes our selected malwares and exploits against target applications. They were selected from a malware collection obtained from honeypots, Web page crawling, and spam traps. It seems that these 57 malwares and exploits have somewhat common exploit techniques. However, we argue that the core merit of anomaly detection system is that, no matter how sophisticated an attack will be, as long as the application's behavior deviates from the baseline, the anomaly can be detected without prior knowledge.

For QQ, we tested 27 password stealer trojans, all of which were detected by Ensemble. Figure 9 shows a representative case. The trojan process (`1180.EXE`) sets a keyboard hook to `QQ.EXE` and tries to log users' keystrokes. The trojan also caused abnormal file accesses: `KERNEL32.DLL` and `ISIGNUP.SYS`. The latter was extracted by the trojan.

We attempted two buffer overflow exploits using the Metasploit framework [6] against Serv-U. Both exploits were detected by Ensemble. One exploit caused ServU to spawn a command line shell, which could be remotely controlled by the attacker. Another exploit

Table 7. Our malware/exploit collection used in false negative evaluation

Target App	# of Malwares/Exploits	Descriptions
Skype	3	Worm
MSN	25	Worm, password trojan
QQ	27	Password trojan
Serv-U	2	Buffer overflow exploits



(a) process dependency

Stack Address	File Pathname
0x157C278F	*PROGRAM_FILES*\Internet Explorer\ Connection Wizard\isignup.sys
0x157C2746	Kernel32.DLL

(b) file read category

Fig. 9. Anomaly detection results of the QQ trojan

Table 8. Anomaly detection results of the Serv-U buffer overflow exploit (unusual file and network access)

Stack Signature(s)	Object Type	Object Name
6607A2DC 6606A17F	File Read	*IE_TEMP*\Content.IE5\H0SBCDN6\putty.exe
112CF1F2 660AC700 660AC7D1	File Write	*IE_TEMP*\Content.IE5\H0SBCDN6\putty.exe
11201534 11211697	File Write	*SYSTEM32*\a.exe
6606A17F 6607A2DC	Dir Read	*IE_TEMP*\Content.IE5\H0SBCDN6\
11211697 11201534	Dir Write	*SYSTEM32*\
660AC7D1 660AC700 112CF1F2	Dir Write	*IE_TEMP*\Content.IE5\H0SBCDN6\
60814BDC 17A77DFF	Connection	193.201.200.66:80 TCP
1B772B23 1B7729D0	IP Prefix	193.201.200.0/23
(Omitted: 106 registry read edges and 26 registry write edges)		

made ServU to download a file and execute it. The exploit was constructed in Metasploit by providing a URL pointing to an executable file (in our experiment, the downloaded executable was `putty.exe`, which was then renamed to `a.exe` and executed). In Table 8, a series of events before the execution of `a.exe` were clearly revealed by failing to match abnormal edges with bipartite graphs in the global profile.

For MSN, we tested 25 worms that hijack MSN to send out malicious contents to the user's contacts. In one example shown in Figure 10, the malware process with a long file name tried to modify registry keys and files that MSN read later.

Skype consists of `Skype.exe` and `SkypePM.exe`. We tested three worms that abused the Skype API to send malicious links to deceive receivers to click them. Since the Skype API on Windows is implemented using the message mechanism, Ensemble detected the worm named `StWinsDat.exe` that sent messages to `Skype.exe`, as shown in Figure 11. Ensemble also detected that Skype read the file `StWinsDat.exe` from two stack addresses that never appeared in the global profile.

As part of the real-deployment in §5.1, we manually executed 25 MSN worms on 3 real machines with different configurations. All abnormal behaviors were detected by Ensemble. Furthermore, it seems that all above anomalies can be covered by the process dependency category. However, we argue that other categories are necessary. For one reason, it is possible that some attacks can happen without process dependency (*e.g.,* anomalies caused by network packets such as Apache-Knacker exploit [3]).

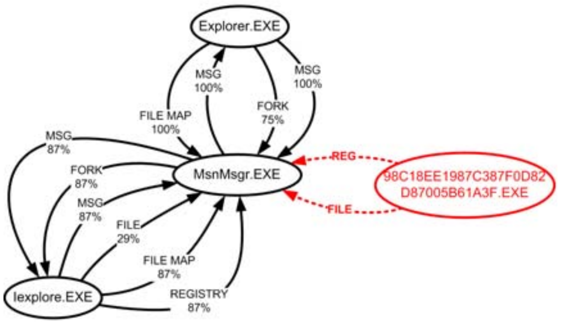
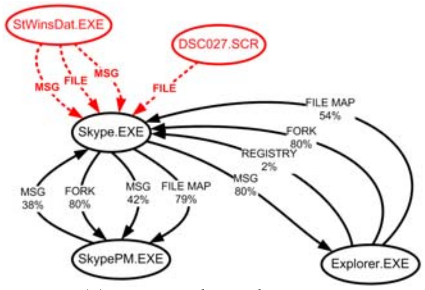


Fig. 10. Anomaly detection results of the MSN worm (process dependency)



(a) process dependency

Stack Address	File Pathname
`0x6C37D084`	*SYSTEM32*\\`stwinsdat.exe`
`0x6C37EFFD`	*SYSTEM32*\\`stwinsdat.exe`

(b) file read category

Fig. 11. Anomaly detection results of the Skype worm

Furthermore, as shown in Figure 9(b), Figure 11(b) and Table 8, other categories provide more detailed information about the anomaly.

5.7 Performance Evaluation

Using four target applications mentioned above, we measured the overhead of our prototype in terms of time and space. The evaluation was done on a commodity Dell Inspiron 530 PC (2.33G Core2 Duo CPU, 2GB memory, with WinXP SP2 installed). We believe that the overall overhead is acceptable. Extra delay incurred by local profile collection is less than 15%. Note that this happens infrequently (*e.g.,* 1 minute per 3 hours), and Ensemble does not collect local profiles for two applications simultaneously. Extra overhead caused by anomaly detection is less than 2%. The logging size of API traces is less than 0.25 MB/min per application. The global profile size is less than 10MB per application. Like software update, the Ensemble server can transfer a "patch" of the new version of the global profile, with a much smaller size.

6 Limitations of Ensemble

While we found Ensemble's approach to be a promising direction for addressing a difficult problem of using run-time profiles for detecting code injections and other run-time anomalies, we also noted limitations that would need to be addressed in the future.

We expect that some applications to be too complex for profiles to converge using limited system-call sampling. Our experiments indicate that this is the case for complex plug-in enabled applications such as IE and MS Word since plug-ins may behave differently from the original applications. Additional sampling and larger communities may help in such cases.

We plan to evaluate Ensemble in a real community with hundreds of users. Privacy concerns must be addressed, even though only summary data about system calls is exchanged with a server.

If a significant fraction of community of users mounted a coordinated attack to pollute the global profile, it is conceivable that the global profile can be corrupted. This is more likely in open communities, where sybil attacks [18] are possible. In closed communities as in enterprise environments, such attacks are much less likely.

Different applications may require different types of profiling. For example, if an application purposely randomizes addresses at function or instruction level (*e.g.,* the network access module of Skype mentioned in §5.5 to obfuscate its behavior), then stack signatures are ineffective. Alternative methods, such as path profiling [15], can be added to handle such applications.

In our design, the stack signature is generated by XORing unique return addresses of stack frames. The probability of collision is non-negligible in 32-bit OS, but very unlikely in 64-bit systems which are becoming increasingly popular.

6.1 Over-Generalization

Each application has a set of "normal behaviors" (true baseline). False negative may happen when the detector-defined normal behaviors go beyond the true baseline (*i.e.,* over-generalized) because the features or methods are not well-chosen or the model is not precise enough (*i.e.,* an imperfect detector). For almost all practical IDS, the detector-defined normal behaviors are broader than the true baseline, thus allowing mimicry attacks. This is a problem with any detectors not just ours. The aggregation process should not introduce much additional over-generalization. Consider the aggregation of local profiles whose diversities are caused by: *(i)* User randomness. Different users can generate different profiles but they mostly fall within true baseline assuming profiles are trusted (User randomness can be regarded as exercising different normal execution paths in the application). *(ii)* System environment randomness. We admit that different system environment may have different set of "normal behaviors". However, this should introduce limited over-generalization, if any at all. In the worst case, we can have separate aggregations/pools for different OSes and software versions as mentioned in §4.2.

6.2 Mimicry Attacks

A perfect detector should leave no opportunity for mimicry attacks which are due to over-generalization. Note that the aggregation process is independent of what features or approaches are used for anomaly detection. The existence of mimicry attack is mainly due to limitations in feature selection and detection techniques, not in profile aggregation. Our focus is to show that with a reasonable detector, how we can reduce false positives rather than making the features rich enough to eliminate the possibility for mimicry attacks.

7 Conclusions

We have described the design of Ensemble, an unsupervised anomaly detection and prevention system relying on a user community to detect or prevent anomalies in popular applications. Local behavioral profiles are combined into a global profile, which can be used to detect or prevent code-injection or behavior-modifying exploits. Hosts participating in Ensemble only need to contribute summary run-time profile data (about 0.5 MB) periodically. Ensemble addresses the problem of merging profiles from hosts that may have different operating environments. From evaluation based on 57 test exploits for four candidate applications, we found that the quality of global profiles, and the resulting false positive rate, significantly improves as the community size grows to approximately 300 users, demonstrating that the use of communities is a practical way to automatically generate behavioral profiles without much manual training, and the resulting behavioral profiles are effective for run-time anomaly detection and prevention.

References

1. Address space layout randomization, `http://blogs.msdn.com/`
2. Application Community, `http://www.darpa.mil/`
3. C. CAN-2003-0245. Apache apr-psprintf memory corruption vulnerability, `http://web.nvd.nist.gov/`
4. Gmail: We're working as a community, give your support!, `http://news.softpedia.com/`
5. McAfee Anti-virus software, `http://mcafee.com/`
6. Metasploit framework, `http://www.metasploit.com`
7. Microsoft Outlook Buffer Overflow in Processing TNEF Messages Lets Remote Users Execute Arbitrary Code, `http://securitytracker.com/`
8. QQ Instant Messenger, `http://im.qq.com`
9. Serv-U FTP Server, `http://www.serv-u.com/`
10. Should we be afraid of Skype, `http://www.ossir.org/`
11. VirusScan Enterprise 8.5i Access Protection rule blocks outbound SMTP mail on Port 25, `https://knowledge.mcafee.com/`
12. Malware flood driving new AV (December 2007), `http://www.infoworld.com/`
13. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the Detection of Anomalous System Call Arguments (2003)
14. Arak, V.: On the worm that affects Skype for Windows users (September 2007), `http://share.skype.com/`
15. Ball, T., Larus, J.: Efficient Path Profiling. In: 29th Annual IEEE/ACM International Symposium on Microarchitecture (1996)
16. Ballardie, T., Crowcroft, J.: Multicast-specific Security Threats and Counter-measures. In: Proc. of the IEEE Symposium on Security and Privacy (1999)

17. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. In: SOSP (2005)
18. Douceur, J.R.: The Sybil Attack. In: Peer-To-Peer Systems: First International Workshop (2002)
19. Ernst, M.: Self-defending software: Collaborative learning for security, http://norfolk.cs.washington.edu/
20. Eskin, E.: Anomarly Detection over Noisy Data using Learned Probability Distributions. In: International Conference on Machine Learning (2000)
21. Eskin, E., Lee, W., Stolfo, S.J.: Modeling system calls for intrusion detection with dynamic window sizes. In: Proceedings of DARPA Information Survivability Conference and Exposition II (DISCEX II) (2001)
22. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly Detection Using Call Stack Information (2003)
23. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for Unix Processes. In: IEEE Symposium on Security and Privacy (1996)
24. Ghosh, A., Wanken, J., Charron, F.: Detecting anomalous and unknown intrusions against programs. In: Proc. of the 1998 Annual Computer Security Applications Conference, ACSAC 1998 (1998)
25. Ghosh, A.K., Schwartzbard, A., Schatz, M.: Learning program behavior profiles for intrusion detection. In: Proceedings of the 1st conference on Workshop on Intrusion Detection and Network Monitoring, vol. 1 (1999)
26. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion detection using sequences of system calls. Journal of Computer Security (1998)
27. Hunt, G., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: Proceedings of the 3rd USENIX Windows NT Symposium (1999)
28. Jon Oberheide, E.C., Jahanian, F.: CloudAV: N-Version Antivirus in the Network Cloud. In: Proceedings of 17th Usenix Security Symposium (2008)
29. King, S.T., Chen, P.M.: Backtracking intrusions. In: SOSP (2003)
30. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Public deployment of cooperative bug isolation. In: Proceedings of the Second International Workshop on Remote Analysis and Measurement of Software Systems, RAMSS (2004)
31. Liblit, B.R.: Cooperative bug isolation. PhD thesis, Berkeley, CA, USA, Chair-Alexander Aiken (2004)
32. Orso, A., Liang, D., Harrold, M.J., Lipton, R.: Gamma system: continuous evolution of software after deployment. SIGSOFT Softw. Eng. Notes 27(4) (2002)
33. Sekar, R., Dhurjati, M.D., Bollineni, P.: A Fast Automation-Based Method for Detecting Anomalous Program Behaviors. In: IEEE Symposium on Security and Privacy (2001)
34. Tucek, J., Newsome, J., Lu, S., Huang, C., Xanthos, S., Brumley, D., Zhou, Y., Song, D.: Sweeper: a lightweight end-to-end system for defending against fast worms. In: EuroSys. (March 2007)
35. Wang, H.J., Platt, J.C., Chen, Y., Zhang, R., Wang, Y.-M.: Automatic misconfiguration troubleshooting with peerpressure. In: OSDI (2004)
36. Warrender, C., Forrest, S., Pearlmutter, B.: Detecting Intrusions using System Calls: Alternative Data Models. In: IEEE Symposium on Security and Privacy (1999)
37. Yeung, D.-Y., Ding, Y.: Host-based intrusion detection using dynamic and static behavioral models. Pattern Recognition 36 (2003)