A framework for behavior-based malware analysis in the cloud

Lorenzo Martignoni[†], Roberto Paleari[‡], and Danilo Bruschi[‡]

Dipartimento di Fisica[†]
Università degli Studi di Udine
lorenzo.martignoni@uniud.it

Dipartimento di Informatica e Comunicazione[‡]
Università degli Studi di Milano
{roberto,bruschi}@security.dico.unimi.it

Abstract. To ease the analysis of potentially malicious programs, dynamic behavior-based techniques have been proposed in the literature. Unfortunately, these techniques often give incomplete results because the execution environments in which they are performed are synthetic and do not faithfully resemble the environments of end-users, the intended targets of the malicious activities. In this paper, we present a new framework for improving behavior-based analysis of suspicious programs. Our framework allows an end-user to delegate security labs, *the cloud*, the execution and the analysis of a program and to force the program to behave as if it were executed directly in the environment of the former. The evaluation demonstrated that the proposed framework allows security labs to improve the completeness of the analysis, by analyzing a piece of malware on behalf of multiple end-users simultaneously, while performing a fine-grained analysis of the behavior of the program with no computational cost for end-users.

1 Introduction

With the development of the underground economy, malicious programs are becoming very profitable products; they are used to spam, to perpetrate web frauds, to steal personal information, and for many other nefarious tasks. An important consequence of this lucrative motivation behind malware development is that these programs are becoming increasingly specialized and difficult to analyze: more and more often they attack very specific classes of users and systems and their code is continuosly updated to introduce additional features and specific modifications to thwart the analysis and eventually evade detection.

To counteract these new threats and to overcome the limitations of traditional malware analysis and detection techniques, security vendors and the research community are moving towards dynamic behavior-based solutions. This approach is becoming the primary method for security labs to *automatically* understand the behaviors that characterize each new piece of malware and to develop the appropriate countermeasures [1–3]. This technology is also used on end-users' hosts, to monitor the execution of suspicious programs and try to detect and block malicious behaviors in real-time [4–6].

Dynamic behavior-based analysis has two major disadvantages: incompleteness and non-negligible run-time overhead. Security labs analyze new malicious

programs automatically in special environments (e.g., virtual machines) which allow very fine grained monitoring of the behavior of the programs. The automatic behavioral analysis of specialized malware becomes more and more difficult because the malicious behaviors manifest only in very specific circumstances. If the behavioral analysis is performed in inappropriate environments, like the synthetic ones used in security labs, the results are very likely to be incomplete. On the other hand, if the malicious program were analyzed directly on an end-user's machine, which is the intended target of the attack, the malicious behavior would have more chances to be triggered and it would be caught as it manifests. Unfortunately, the strict lightweight constraint required for end-users' systems does not allow a fine grained analysis of the behaviors of the programs [2, 3]. Consequently, some malicious behaviors (e.g., the leakage of sensitive information) cannot be detected on end-users' machines. Current solutions address the incompleteness of dynamic analysis by systematically exploring all environment-dependent programs paths [7, 8].

In this paper we propose a new framework for supporting dynamic behavior-based malware analysis, based on cloud computing, that blends together the computational power available in security labs (the cloud) with the heterogeneity of end-users' environments. The rationale of the framework are the two following assumptions. First, the security lab has no limit on the computational resources available and can exploit hardware features, in combination with recent advances in research, to further improve its computational capabilities [9–11]. Second, end-users' environments are more realistic and heterogeneous than the synthetic environments typically available in security labs and consequently are better suited for analyzing potentially malicious software. The proposed framework allows an end-user to delegate a security lab the execution and the analysis of a potentially malicious program and to force the program to behave as if it were executed directly in the environment of the former. The advantage is twofold. It allows the security lab to monitor the execution of a potentially malicious program in a *realistic end-user's environment* and it allows end-users to raise their level of protection by leveraging the computational resources of the security lab for fine-grained analysis that would not be feasible otherwise. Since each end-user's environment differs from the others and since the behavior of a program largely depends on the execution environment, through our framework, the security lab can improve the completeness of the analysis by observing how a program behaves in *multiple realistic end-users' environments*. Such in the cloud execution is made possible by a mechanism we have developed for forwarding and executing (a subset of) the system calls invoked by the analyzed program to a remote end-user's environment and for receiving back the result of the computation. As the execution path of a program entirely depends on the output of the invoked system calls, the analyzed program running in the security lab behaves as if it were executed directly in the environment of the user.

To evaluate the proposed approach, we have implemented a prototype for Microsoft Windows XP. Our evaluation witnessed that the distributed execution of programs is possible and the computational impact on end-users is negligible. With respect to the traditional analysis in the security lab, the analysis of

malicious programs in multiple execution environments resulted in a significant relative improvement of the code coverage: with just four additional distinct end-users' environments we achieved an improvement of ∼15%.

To summarize, the paper makes the following contributions: (I) a new framework for dynamic behavior-based malware analysis in the cloud; (II) a working prototype of the above mentioned framework, that has also been integrated into an existing behavior-based malware detector; (III) an evaluation of the proposed framework, demonstrating the feasibility and the efficacy of our idea.

2 Overview

Imagine a malicious program, like the one shown in Fig. 1, that resembles the behavior of the BANCOS malware [12]. To ease the presentation we use high-level APIs of Microsoft Windows; nevertheless our approach works directly with the system calls invoked by these functions. The program polls the foreground window to check whether the user is visiting the website of a Brazilian bank. The existence of such a window is the *trigger condition* of the malicious behavior. If the bank website is visited, the program displays a fake authentication form to tempt the user to type his login and password. Finally, the program forwards the stolen credentials to a remote site.

The automatic analysis of such a piece of malware in a *synthetic execution environment*, like those available in a security lab, is very likely to give incomplete results. Such an environment is generated artificially and consequently it cannot satisfy all the possible trigger conditions of malicious programs. Furthermore, some malicious programs expect inputs from the user and then behave accordingly. As the analysis is performed automatically, user inputs are also artificial and that can prevent the triggering of certain behaviors. On the other hand, we have *realistic execution environments*, the systems of the end-users, which are more suited for analyzing a piece of malware like BANCOS, as they are the intended victims of the malicious activity. Indeed, in the system of a certain class of users, the users of Brazilian banks, our sample malicious program would manifest all its behaviors. Unfortunately, although such systems are more suited for the analysis, it is not reasonable to expect to use all their resources for detecting and stopping potentially malicious programs (fine grained analysis can introduce a slowdown by a factor of 20 [3, 13]). Consequently, host-based detectors perform only very lightweight analysis and cannot detect certain malicious behaviors (e.g., to detect that sensitive information are being leaked using data-flow analysis).

2.1 Delegating the analysis to the cloud

In our framework the behavior-based analysis of a new suspicious program is performed in the cloud: the user U does not run directly on his system the suspicious program, nor the malware detector, but he requests the security lab L to analyze the program on his behalf; in turn the latter requests the help of the former to mitigate the fact that its execution environment is synthetic. Our approach to overcome the limitations of the execution environment of L is based on

```
VirtualAlloc();
...
VirtualFree();
while (true) {
  hwnd1 = GetForegroundWindow();
  title = GetWindowText(hwnd1);

  if (title == "Banco do Brasil" ||
      title == "Banco Itau" || ...) {
    // Display a fake login screen for
    // the site
    hwnd2 = CreateWindow(...);
    ...
    // Send credentials to a remote site
    socket = WSAConnect();
    WSASend(socket, ...);
    ...
    break;
  }

  Sleep(500);
}
```

Fig. 1. Pseudo-code of a sample malicious program that resembles the BANCOS trojan.

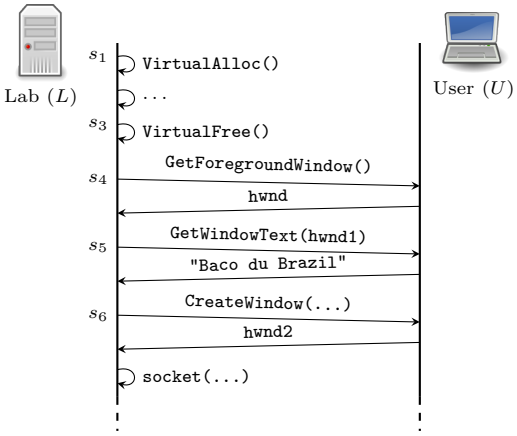

Fig. 2. Diagram of the execution of the sample malicious program in the security lab (L), by forcing the program to behave as in the environment of the end-user (U).

the following assumption: a program interacts with the environment by invoking system calls, and the execution path taken by the program entirely depends on the output of these calls [14]. In our particular context, this assumption means that the triggering of a malicious behavior entirely depends on the output of the system calls invoked. It follows that, to achieve our goal, it is sufficient to force the system calls executed by the program in L to behave as they were executed in U. To do that, the system calls, instead of being executed in L, are executed in U, and L simulates their execution by using the output produced by U. It is worth noting that only a small subset of all the system calls executed by the program might actually affect the triggering of a malicious behavior. Examples of such system calls are (I) those used to access user's data (e.g., the file system and the registry), (II) those used to query particular system information (e.g., active processes, system configuration, open windows), and (III) those used to interact with the users (e.g., to process keyboard and mouse events). Therefore, the collaboration of U is needed only for these system calls, while the remaining ones can be executed directly in L.

Fig. 2 shows how our sample malicious program is executed and analyzed leveraging our framework. The scenario of the analysis is the following. The user U has received a copy of the program by email (or by another vector) and he executes the program. With a conventional behavior-based detector the program would be analyzed entirely on the host. With our framework instead, the program is not executed locally but it is submitted to the security lab L, that executes and analyzes the program with the cooperation of the user. The new analysis environment thus becomes $\langle L, U \rangle$. All the system calls executed by the program are intercepted. Our sample program initially executes some system calls $s_1, \ldots, s_3$ whose output does not depend on the environment (e.g., to allocate memory). These system calls are executed directly in L. Subsequently,

the program tries to detect whether the user is browsing a certain website: it invokes $s_4 =$ `GetForegroundWindow` to get a reference to the window currently active on the desktop of the user. As the output of this call highly depends on the execution environment, L requests U to execute the call: L forwards s_4 to U, U executes s_4 and sends back the output to L. The program does not notice what is happening in the background and continues the execution. The next system call is $s_5 =$ `GetWindowText`, which is used to get the title of the foreground window. As one of its input arguments (`hwnd1`) is the output of a system call previously executed in U, s_5 is also executed in U. Supposing that the user in U is actually visiting a website targeted by the program, the trigger condition is satisfied and the program displays the fake login form to steal the user's credentials. As this activity involves an interaction with the user and such interaction is essential to observe the complete behavior of the program, the system calls involved with this activity are also forwarded to U, to get a realistic input. L can eventually detect that there is an illegitimate information leakage.

The in the cloud execution of a potentially malicious program does not expose the end-user to extra security risks. First, we confine the dangerous modifications the program could make to the system in the environment of the security lab. Second, more malicious behaviors can be detected and stopped, because the analysis performed in the lab is more thorough. Third, the execution of the program consumes less resources, as the user is in charge of executing a subset of all the system calls of the program. Forth, annoying popups are still redirected and shown to the user, but that would happen also if the program were executed normally.

2.2 Exploiting diversity of end-users' environments

The proposed framework allows to monitor the execution of a potentially malicious program in multiple execution environments. Given the fact that end-users' environments are very heterogeneous (e.g., users use different software with different configurations, visit different web-sites), it is reasonable to expect that the completeness of the analysis improves with the increase of the number of different environments used.

To analyze a program in multiple execution environments, it is sufficient to run multiple instances of the analyzer, $L_1, \ldots, L_n$, such that each instance cooperates with a different environment $U_1, \ldots, U_n$ to execute the system calls that might affect the triggering of the malicious behaviors (i.e., the environments used are those of n of the potential victims of the malicious program, chosen according to some criteria). The security lab can thus observe how each analysis environment $\langle L_i, U_i \rangle$ affects the behavior of the program and can merge and correlate the behaviors observed in each execution.

Fig. 3 shows how the analysis of our sample program is performed simultaneously in multiple execution environments $\langle L_1, U_1 \rangle, \ldots, \langle L_6, U_6 \rangle$. Each execution is completely independent from the others but the results of the analysis are collected and correlated centrally by L. As $U_1, \ldots, U_6$ are distinct environments, we expect the forwarded system calls to produce different output (e.g., to return

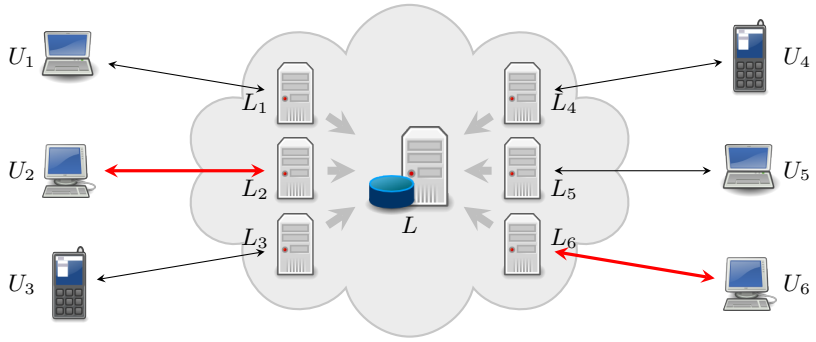

Fig. 3. Diagram of the execution of multiple instances of the analysis of a suspicious program in multiple execution environments $\langle L_1, U_1\rangle, \ldots, \langle L_6, U_6\rangle$. The central entity L aggregates the results of each analysis.

different window titles) and thus to cause the various instances of the analyzed program to follow different paths. In the example, we have that the trigger condition is satisfied only in U_2 and U_6, but the web sites being visited are different (one user is visiting the web site of "Bancos do Brazil" and the other one the web site of "Banco Itau"). Therefore, the correlation of the results reveals that the program is effectively malicious and some of its trigger conditions.

3 Design and implementation

The two parties participating to the in the cloud analysis of a program are the security lab, L for short, and the end-user (the potential victim), U for short. In this section we describe the components we have developed for these two parties to make such distributed execution possible. The current prototype implementation is specific for Microsoft Windows XP, but the support for other versions of the OS can be added with minimal efforts. At the moment, our prototype can successfully handle all the system calls involving the following system resources: file, registry keys, system and processes information, and some graphical resources.

3.1 Executing a program in multiple environments

System calls hooking. To intercept the system calls executed by the analyzed program, we leverage a standard user-space hooking technique. We start the process we want to monitor in a suspended state and then inject a DLL into its virtual address space. The DLL hooks the functions `KiIntSystemCall` and `KiFastSystemCall`, two small function stubs used by Microsoft Windows for executing system calls [15, 16]. This approach allowed to simplify the development and facilitated the integration of the framework into an existing malware detector.

System calls proxying. A user-space application cannot directly access the data structure representing a particular resource of the system (e.g., a file, a registry key, a mutex, a window) but it has to invoke the appropriate system calls to obtain an opaque reference, a *handle*, to the resource and to manipulate it. We exploit this characteristic of the operating system to guarantee a correct functioning of the analyzed program, and to simulate the existence of resources with certain properties that exists on a remote system, but do not in the system in which the program is executed. When a system call is invoked, we analyze the type of the call and its arguments to decide how to execute it: locally or remotely.

To differentiate between *local* and *remote* calls, we check if the system call creates a handle or if it uses a handle. To create a handle means to open an existing resource or creating a new one (e.g., to open a file), while to use a handle means to manipulate the resource (e.g., to read data from a open file). In the first case, we analyze the resource that is being opened and according to some rules (details follow) we decide whether the manipulation of the resource might influence the triggering of a malicious behavior. If not, we consider the resource and the system call *local* and we execute the call in L. Otherwise, we consider the resource and the system call *remote* and we forward and execute the latter in U. When we intercept a system call that uses a handle, we check whether the resource being manipulated (identified by the handle) is local or remote and we execute the call in L or U accordingly.

Fig. 4 represents the various components we have developed (highlighted) to intercept system calls and to execute them either locally or remotely. All system calls executed by the analyzed program P are intercepted. Local system calls are passed to the kernel as is, remote ones are forwarded to the system of the end-user. To execute a remote syscall in U, L serializes the arguments of the system call and sends them to U. The receiver deserializes the arguments, prepares the program state for the execution (i.e., by setting up the stack and the registers), and then executes the call. When the syscall returns, U serializes the output arguments and sends them back to L. Finally, L deserializes the output arguments, where the program expects them, and resumes the normal execution. The program P cannot notice when a system call is executed elsewhere, because it finds in memory the expected output.

On paper, the mechanism for serializing and proxying a system call looks simple; however, its implementation is very challenging. The Microsoft Windows system call interface, known as *native API*, is poorly documented. We put a lot of reverse engineering efforts to understand how to properly serialize all system calls and their arguments. After all, the Windows native API turned out to be well suited for proxying and to simulate the existence of resources that physically reside on a different system. No system call can operate concurrently on two resources, resources can always be distinguished, and system calls manipulating the same resource are always executed in the same environment.

Choosing remote system calls. Remote system calls are selected using a whitelist. The whitelist contains a list of system calls names and a set of con-

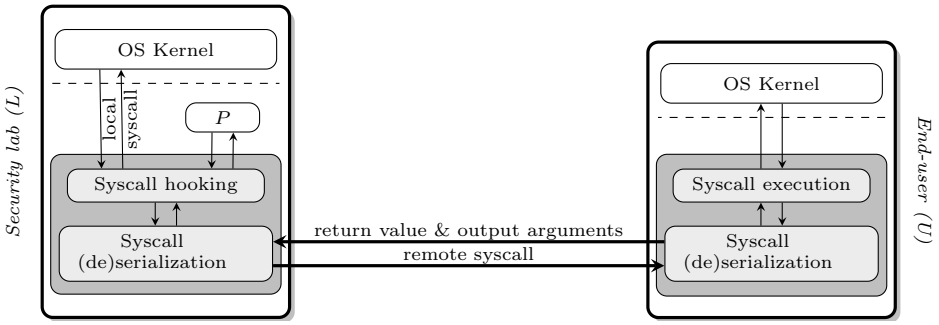

Fig. 4. System calls interception and remote execution (P is the analyzed program)

ditions on the arguments. Examples of the system calls we consider remote are: `NtOpenKey`, `NtCreateKey` (if the arguments indicate that the key is being opened for reading), `NtOpenFile`, `NtCreateFile` (if the arguments indicate that the file is being opened for reading), `NtQuerySystemInformation`, and `NtQueryPerformanceCounter`. The handles returned by these calls are flagged as remote, by setting the most significant bits (which are unused). Thus, we can identify subsequent system calls that access a remote resource and we have the guarantee that no overlap between handles referencing local and remote resources can occur.

GUI system calls. User's inputs and GUI resources often represent trigger conditions. For this reason it is important to let the analyzed program to interact with realistic user's inputs (i.e., GUI events) and resources. Although in Microsoft Windows all the primitives of the graphical user interfaces are normal system calls, to facilitate the proxying, we rely on Windows Terminal Services subsystem to automatically forward the user interface of the monitored application from the lab to the user's machine. In particular, our prototype uses *seamless* RDP (Remote Desktop Protocol) [17], that allows to export to a remote host the graphical interface of a single application instead of the entire desktop session. Therefore, if the analyzed program executed in the lab displays the user a fake login form and blocks for inputs, the form is transparently displayed in U and the received user's events (keystrokes and mouse clicks) are sent back to the program running in L.

The solution based on RDP allows only to forward a GUI to a remote system. However, the session in which the application is run belongs to L. Thus, attempts to query the execution environment would return the status of the environment in L. As an example let us consider the system calls associated with the API functions `GetForegroundWindow` and `GetWindowText`, used by our sample malware (Fig. 1) to check if the victim is visiting the website of a Brazilian bank. Without any special handling these system calls would return the windows of the session (on L). We want instead these calls to return information about the windows found in the remote environment. To do that, we execute them remotely as any other remote system call.

One-way isolation. One of the goal of our framework is to protect the system of the end-user from damages that could be caused by the analyzed program, without interfering with the execution of the program. The approach we adopt to achieve this goal is based on one-way isolation [18]: "read" accesses to remote system resources are allowed, but "write" accesses are not and are performed locally. That is, if the program executes a system call to create or to modify a resource we normally consider remote, we treat the resource as local and do not proxy the call. To guarantee a consistent program state, we also execute locally all subsequent system calls involving such resource.

In case the analyzed program turned out to be benign, system changes made in the lab environment could be committed to end-user's environment. Our prototype currently does not support this feature, nor does it support the correct isolation of a program that accesses a resource that is concurrently accessed by another.

3.2 An in the cloud behavior-based malware detector

In order to demonstrate how our framework can naturally complement behavior-based malware detectors, we have integrated it in an existing detector [2], which is based on virtual machine introspection and is capable of performing fine grained information flow tracking and to identify data-flow dependencies between system calls arguments. The malware detector is built on top of a customized system emulator, which supports system calls interception and taint analysis with multiple taint labels. As our framework works directly inside the guest, the integration of the two components required only a trivial modification to allow the detector to isolate the system calls executed by the suspicious program from those executed by our prototype to proxy system calls and to ignore the latter.

To monitor the execution of a suspicious program in multiple end-users' environments it is sufficient to run multiple instances of the enhanced malware detector just described, where each instance collaborates with a different end-user's machine, and to merge the results. We have not yet addressed the problem of correlating the results of multiple analyses.

4 Evaluation

This section presents the results of testing our prototype implementation of the framework and presents a conceptual comparison of our approach with existing solutions that try to systematically explore all program paths. We evaluated the prototype with benign and malicious programs. The results of the evaluation on benign programs witness that our approach does not interfere with normal program execution and that it introduces a negligible overhead. Moreover, the evaluation demonstrates that the analysis of a piece of malware in multiple execution environments significantly improves the completeness of the results: with the collaboration of *just four* different execution environments we observed a $\sim$15% relative improvement of the code coverage.

Program	Action	Local	Remote
ClamAV	Scan (remote) files with (remote) signatures	166,539	1,238
Eudora	Access and query (remote) address book	1,418,162	11,411
Gzip	Compress (remote) files	19,715	93
MS IE	Open a (remote) HTML document	1,263,385	10,260
MS Paint	Browse, open, and edit (remote) pictures	1,177,818	9,708
Netcat	Transfer (remote) files to another host	16,007	93
Notepad	Browse, open, and edit (remote) text files	929,191	7,598
RegEdit	Browse, view, and edit (remote) registry keys	1,573,995	13,697
Task Mgr.	List (remote) running processes	33,339	241
WinRAR	Decompress (remote) files	71,195	572

Table 1. List of tested benign programs, actions over which each program was exercised, and number of locally and remotely executed system calls (GUI system calls are not counted).

Experimental setup. The infrastructure used for the evaluation corresponds to the one described in Section 3.2, with the difference that, instead of performing behavior-based detection, we tracked the basic blocks executed in each run of the experiments. To simulate the lab environment we used a vanilla installation of Windows XP running inside the emulator, while as users' environments we used some other machines and we acted as the end-users.

Evaluation on benign programs. To verify that our framework did not interfere with the correct execution of the programs, we executed through our prototype multiple benign applications. The tested programs included both command line utilities and complex GUI applications. Table 1 reports the set of programs tested, together with the actions over which each program was exercised and with the number of local and remote system calls. We interacted with each program to perform the operations reported in the table. As we ran the experiments with the proxying of all supported system calls enabled, the numbers in the table indicate the total number of remotely executed calls and not only those involved with the described actions. For example, we used ClamAV to scan all the content of a directory. Through our framework the anti-virus transparently scanned a directory existing only in the simulated end-user's system, using a database of signatures which also existed only in the remote system.

We successfully executed all the actions reported in the table and verified that the resources that were accessed effectively corresponded to those residing on the system of the end-users. The number of system calls executed indicates that the programs used for the evaluation are quite complex and thus that our results are good representatives. We can conclude that: (I) system calls accessing remote resources do not interfere with system calls accessing local resources, (II) our framework does not interfere with the correct execution of programs, and (III) system calls proxying allows to transparently access system resources residing on remote hosts.

Performance overhead. We used a subset of the benign programs of Table 1 to evaluate the overhead introduced by our framework on the systems of the

user and of the security lab. We observed that the number of remotely executed system calls depended on the type of applications and the actions exercised; consequently the overhead depended on these factors. On the system of the end-user, we measured a CPU, memory, and network usage that was roughly proportional to the number of remotely executed system calls. Nevertheless, in all cases, the resources consumed never exceeded the resources consumed when the same programs were executed natively on the system: on average we observed a 60% and 80% reduction of CPU and memory usage respectively. On the other hand, we noticed a slight increase of the resource usage in the system in the lab: on average we observed a 36% and 77% increase of CPU and memory usage respectively. We also measured that, on average, 956 bytes have to be transferred over the network to remotely execute a system call. For example, the execution of RegEdit required in total to transfer 1030Kb of data. In conclusion, our framework has negligible performance impact on the end-user and the impact on the security lab, without considering the overhead introduced by the analysis run on the framework, is sustainable and can be drastically reduced by improving the implementation (e.g., by compressing data before transmission).

Evaluation on malicious programs. We evaluated our framework against multiple malicious programs representing some of the most common and recent malware families. The goal of the evaluation was to measure whether the analysis of multiple executions of the same piece of malware, in different end-users' environments, gives more complete results then the analysis of a single execution of the program in an unrealistic environment (i.e., the vanilla installation of Windows XP).

To quantify the completeness of the results we measured the increase of code coverage. We initially executed batch each malicious program in the environment of the security lab and we recorded the set of unique basic blocks executed (excluding library code). Subsequently, we ran each malicious program multiple times through our prototype, each time in collaboration with a different end-user's environment, and again we recorded the set of unique basic blocks executed. Therefore, if b_0 represents the set of basic blocks executed in the environment of the security lab, and $b_i, i > 0$, represents the set of basic blocks executed with the collaboration of the i^{th} end-user's environment, the increase of code coverage after the i^{th} execution is measured as $|b_i \setminus (b_{i-1} \cup ... \cup b_0)|$.

Fig. 5 reports the relative increase of code coverage (using b_0 as baseline) measured during our evaluation, leveraging just four different end-users' environments and 27 different malware samples. The figure clearly shows that in the majority of the cases we have a noticeable relative increase of the code coverage; the average increase is 14.53%, with a minimum of 0.24%, to a maximum of 60.92%. It is worth noting that, although the observed improvements appear minimal, most of the time small percentages correspond to the execution of hundreds of new basic blocks. It is also important to note that certain environments contributed to improve the results with certain malware but did not contribute at all with others. Indeed, the four environments contribute respectively on average 25.35%, 30.86%, 18.14%, and 25.68% of the total increase observed. For

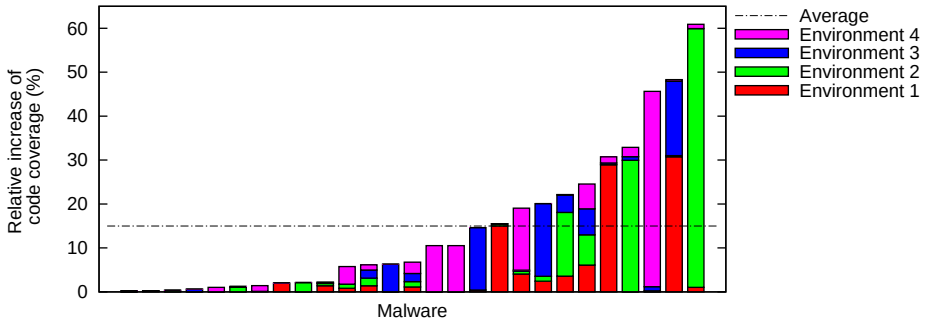

Fig. 5. Relative increase of code coverage obtained by analyzing the tested malware samples in multiple execution environments.

example, during the analysis of a variant of SATILOLER, we noticed that the monitoring of web activities was triggered only in one of the four environments, when we visited a particular website. Thus, in this environment we observed a 16.54% increase of the relative code coverage, corresponding to the execution of about 140 new unique basic blocks; the observed increase in the other environments did not exceed 3%.

In conclusion, we believe the relative improvements observed during the evaluation testify the effectiveness of the proposed approach at enhancing the completeness of dynamic analysis.

Conceptual comparison with input oblivious analyzers. Input oblivious analyzers are tools capable of analyzing exhaustively a malicious program by systematically forcing the execution of all program paths [7, 8]. When an input-dependent control flow decisions is encountered, both program branches are explored. Such systematic exploration is achieved by manipulating the inputs and updating the state of the program accordingly, leveraging constraint solvers, to force the execution of one path and then of the other.

The framework we propose in this paper addresses the same problem through a completely different approach. Although our methodology might appear less systematic, it has the advantage that, by leveraging real execution environments, it can deal with complex trigger conditions that could exhaust the resources of input oblivious analyzers. For example, trigger conditions dominated by a complex program structure might easily generate an unmanageable number of paths to explore and unsolvable constraints. Indeed, several situations are already known to thwart these systems [19, 20]. Examples of other situations that can easily render input oblivious analyzers ineffective are malicious programs with payload delivered on-demand (e.g., the Conficker malware [21]) and programs with hidden malicious functionality, like rouge anti-viruses, where the trigger conditions consist in multiple complex asynchronous events. As we assume that sooner or later the malicious program will start to reap victims, we can just sit and watch what a program does in each victim's system, without being affected by the complexity of trigger conditions. At the first sign of malicious activity, we con-

sider the program as malicious; then we can notify all victims, but we could also continue to analyze the program in some of the affected systems.

5 Discussion

Privacy and security issues. The framework we propose can clearly raise privacy issues: by controlling the system calls executed on the systems of an end-user, the security lab can access sensitive user's data (e.g., files, registry keys, GUI events). We are convinced that, considering the current trend, the privacy issues introduced by our approach are comparable to already existing issues. For example, commercial behavior-based detectors incorporate functionality, typically enabled by default, to submit to labs suspicious executables or memory dumps of suspicious processes (which can contain sensitive user data). Thus privacy of users is already compromised. Moreover, the security lab is just a special provider of cloud services: users have to trust it like they trust other providers (e.g., email providers and web storage services).

Detection and evasion. Our framework is sensitive to various forms of detection and evasion. To prevent evasion attacks based on the identification of emulated or analysis environments, it would be sufficient to build our framework on top of undetectable systems for malware analysis [22]. The limitations of our current implementation (e.g., lack of support for inter-process communication) can also offer opportunities for detection and evasion. We believe the majority of the attacks will not be possible with a complete implementation.

6 Related work

Malware analysis in the cloud. CloudAV is the first implementation of an in the cloud malware detector through which end-users delegate to a central authority the task of detecting if an unknown program is malicious or not [23]. A similar approach, called "collective intelligence", has also been introduced in a commercial malware detector [24]. Such centralized detection gives two major benefits. First, the analysis no longer impacts on end-users' systems, and being centralized, it can be made more fine-grained. For example CloudAV analyzes programs simultaneously, with multiple off-the-shelf detectors. Second, the results of the analysis can be cached to serve future requests of other users at no cost. This paper further enhances these existing solutions by proposing a framework that leverages the systems of potential victims for making the behavioral analysis much more complete.

Behavior-based malware analysis. Our proposed solution is not a malware detector, but is rather a framework that enhances the capabilities of existing dynamic behavior-based detectors. Examples of malware detectors that could integrate our approach are TTAnalyze [1], Panorama [3], CWSandbox [25], and [2]. The problem of the incompleteness of dynamic approaches for malware analysis has been addressed by Moser et al. and Brumley et al. [7, 8]. Both systems allow the automatic exploration of multiple execution paths. A thorough comparison between these systems and ours is presented in Section 4.

Sandboxed programs execution. Sun et al. introduced a one-way isolation technique to safely execute untrusted programs [18]. Their approach consists in isolating the effects of an untrusted program from the rest of the system by intercepting system calls that modify the file-system and redirecting them to a cache, invisible to other processes. When the untrusted program terminates, the user can choose to discard these modifications, or to commit them to the real system. The approach we adopt to proxy the access to remote system resources is similar to the one proposed by Sun et al.

Remote system call execution. Remote system call execution has been successfully used to implement a high-throughput computation environment based on Condor [26], where files stored on remote nodes of the environment are made accessible locally and transparently by proxying the appropriate system calls. Similarly, the $\mathcal{V}^2$ project [27] includes support for remote system call execution. Our framework adopts the same strategy, but leverages system call proxying to achieve a completely different goal.

7 Conclusion

In this paper, we presented a framework that enables sophisticated behavior-based analysis of suspicious programs in multiple realistic and heterogeneous environments. We achieve this goal by distributing the execution of the program between the security lab (with unlimited computational resources) and the environments of potential victims of the program (which are heterogeneous by definition and might affect differently the behavior of the analyzed program), by forwarding to the latter certain system calls. We have implemented an experimental prototype to validate our idea and integrated it into an existing behavior-based malware detector. Our evaluation demonstrated the feasibility of the proposed approach, that the overhead introduced is very small, and that the analysis of multiple execution traces of the same malware sample in multiple end-users' environments can improve the results of the analysis.

References

1. Bayer, U., Kruegel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: Proceedings of the Annual Conference of the European Institute for Computer Antivirus Research. (2006)
2. Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., Mitchell, J.C.: A Layered Architecture for Detecting Malicious Behaviors. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection. (2008)
3. Yin, H., Song, D., Egele, M., Kirda, E., Kruegel, C.: Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In: Proceedings of the Conference on Computer and Communications Security. (2007)
4. NovaShield: `http://www.novashield.com/`.
5. Panda Security: True Prevent `http://research.pandasecurity.com/archive/How-TruPrevent-Works-_2800_I_2900_.aspx`.
6. Sana Security: `http://www.sanasecurity.com/`.

7. Moser, A., Kruegel, C., Kirda, E.: Exploring Multiple Execution Paths for Malware Analysis. In: Proceeding of the IEEE Symposium on Security and Privacy. (2007)
8. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., Yin, H.: Towards Automatically Identifying Trigger-based Behavior in Malware using Symbolic Execution and Binary Analysis. Technical Report CMU-CS-07-105, Carnegie Mellon University (2007)
9. Chabbi, M.: Efficient Taint Analysis Using Multicore Machines. Master's thesis, University of Arizona (2007)
10. Nightingale, E.B., Peek, D., Chen, P.M., Flinn, J.: Parallelizing security checks on commodity hardware. In: Proceedings of the international Conference on Architectural Support for Programming Languages and Operating Systems. (2008)
11. Ho, A., Fetterman, M., Clark, C., Warfield, A., Hand, S.: Practical Taint-based Protection Using Demand Emulation. In: Proceedings of the EuroSys Conference. (2006)
12. F-Secure: Trojan Information Pages: Bancos.VE `http://www.f-secure.com/v-descs/bancos_ve.shtml`.
13. NoAH Consortium: Containment environment design. Technical report, European Network of Affined Honeypots (2006)
14. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A Secure Environment for Untrusted Helper Applications. In: Proceedings of the USENIX Security Symposium. (1996)
15. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. Addison-Wesley (2006)
16. Russinovich, M., Solomon, D.: Microsoft Windows Internals. 4th edn. Microsoft Press (2004)
17. Cendio: SeamlessRDP – Seamless Windows Support for rdesktop `http://www.cendio.com/seamlessrdp/`.
18. Sun, W., Liang, Z., Sekar, R., Venkatakrishnan, V.N.: One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In: Proceedings of the Symposium on Network and Distributed Systems Security. (2005)
19. L. Cavallaro and P. Saxena and R. Sekar: On the Limits of Information Flow Techniques for Malware Analysis and Containment. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment. (2008)
20. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Impeding Malware Analysis Using Conditional Code Obfuscation. In: Proceedings of the Annual Network and Distributed System Security Symposium. (2008)
21. Porras, P., Saidi, H., Yegneswaran, V.: An Analysis of Conficker's Logic and Rendezvous Points. Technical report, SRI International (2009)
22. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware Analysis via Hardware Virtualization Extensions. In: Proceedings of the Conference on Computer and communications security. (2008)
23. Oberheide, J., Cooke, E., Jahanian, F.: CloudAV: N-Version Antivirus in the Network Cloud. In: Proceedings of the USENIX Security Symposium. (2008)
24. Panda Security: From Traditional Antivirus to Collective Intelligence (2007)
25. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using CWSandbox. IEEE Security and Privacy (2007)
26. Livny, M., Basney, J., Raman, R., Tannenbaum, T.: Mechanisms for High Throughput Computing. SPEEDUP Journal (1997)
27. VirtualSquare: Remote System Call `http://wiki.virtualsquare.org/index.php/Remote_System_Call`.