CloudSylla: Detecting Suspicious System Calls in the Cloud

Marc Kührer, Johannes Hoffmann, and Thorsten Holz
{firstname.lastname}@ruhr-uni-bochum.de

Horst Görtz Institute for IT-Security, Ruhr-University Bochum, Germany

Abstract. To protect computer systems against the tremendous number of daily malware threats, security software is typically installed on individual end hosts and the responsibility to keep this software updated is often assigned to (inexperienced) users. A critical drawback of this strategy, especially in enterprise networks, is that a single unprotected client system might lead to severe attacks such as industrial espionage. To overcome this problem, a potential approach is to move the responsibility to utilize the latest detection mechanisms to a centralized, continuously maintained network service to identify suspicious behavior on end hosts and perform adequate actions once a client invokes malicious activities. In this paper, we propose a security approach called *CloudSylla* (*Cloud*-based *SY*sca*LL A*nalysis) in which we utilize a centralized network service to analyze the clients' activities directly at the API and system call level. This enables, among other advantages, a centralized management of signatures and a unified security policy. To evaluate the applicability of our approach, we implemented prototypes for desktop computers and mobile devices and found this approach to be applicable in practice as no substantial limitations of usability are caused on the client side.

1 Introduction

Malicious software needs to invoke API, respectively, system calls to cause substantial damage, thus monitoring these calls is a promising approach for detecting suspicious activities [1, 2]. Consequently, this technique is often adopted by security and malware protection services, which are typically deployed locally on end hosts. The drawback of this strategy is that each client is responsible for keeping its security software updated in short-time intervals to also detect latest zero-day attacks. When the software is not updated on a regular basis, the host might somehow get infected with malware. Especially in large-scale networks, this is a severe problem since an infected client machine might be used as an entrance point for more substantial attacks such as industrial espionage.

A reasonable approach is to move the identification of malicious activities to a centralized and more powerful network service. In the past, several approaches were proposed [3–5], in which the actual analysis process is performed in the Cloud. The clients are then no longer required to keep their detection mechanisms updated continuously, reducing the amount of required computing power on end hosts significantly—particularly important for mobile devices with limited power capabilities. Nevertheless, all these approaches operate on a rather coarse-grained level, e.g., CloudAV [3] only analyzes whether executables are detected by antivirus engines which might fail for obfuscated malware. To perform a more fine-granular inspection of the clients' behavior in the Cloud, we introduce an analysis mechanism that operates directly on API and system calls invoked on the end hosts. Outsourcing the inspection of these operations to a Cloud implies several benefits, yet might also induce serious drawbacks. To evaluate the applicability of our security mechanism, we implemented prototypes for desktop computers using Windows and mobile devices using Android and find our approach to efficiently detect malicious activities on the end hosts by analyzing invoked API and system calls at a centralized network service.

In summary, this paper makes the following contributions:

– We propose an approach to move the detection of malicious behavior from individual end hosts to a centralized network service. To perform a fine-granular inspection of end host activities, we analyze the corresponding API and system calls in the Cloud to determine if these activities are malicious.
– We implemented prototypes for desktop computers and mobile devices to monitor and forward invoked API and system calls to the Cloud service.
– In empirical evaluations, we demonstrate the feasibility of our approach. The typical runtime overhead of our implementation is negligible for already known applications due to efficient caching mechanisms. New and therefore unknown applications can still be analyzed in a satisfying amount of time.

2 General Approach

A fine-granular approach to improve the clients' security, particularly applicable in enterprise networks with good connectivity and low latency, is to outsource local malware detection to a less vulnerable and more powerful Cloud service that identifies malicious activities by inspecting API and system calls—both referred to as *syscall* in the following although we focus on API calls in our Windows prototype. This Cloud-based strategy reduces the administrative overhead significantly, since end hosts are no longer required to maintain local detection mechanisms and keep signatures updated in short-time intervals. Updating detection mechanisms can be accomplished more easily as changes need to be performed on the Cloud side only, which enables a unified security policy. This centralized analysis also enables a correlation of the behavior of all hosts that send data to the Cloud service, enabling detection mechanisms like BotMiner [6].

To detect malicious activities, we require each end host to forward specific events at the API and system call level to the Cloud and await approval or denial to perform these actions locally. More specific, once a syscall is invoked by a client process, the corresponding syscall arguments (e.g., filenames and URLs) are individually looked up in locally stored caching instances, containing information for trusted, malicious, and analyzed but unsuspicious values. If not cached, the syscall including the arguments is forwarded to the Cloud. The Cloud first applies signatures matching, i.e., probes if the syscall is part of a signature, a sequence of consecutively invoked syscalls. Afterwards, the individual arguments are checked against blacklists and looked up at external sources. If no argument is found to indicate malicious behavior, the syscall is executed on the end host. If malicious behavior is identified, the end host terminates the malicious application or, more restrictive, is automatically blocked from accessing critical infrastructure such as the local network, depending on a specifiable local security policy.

Selecting a reasonable set of syscalls to monitor is a critical but necessary task to reduce the overall number of analysis requests forwarded to the Cloud. Monitoring irrelevant syscalls wastes network bandwidth and execution time of the clients, however, tracking an insufficient set of syscalls might miss important activities to detect malicious behavior. Modern operating systems provide a large number of syscalls, and in some cases, multiple syscalls perform almost the same operation (e.g., creating a process). We thus need to find basic syscalls (e.g., `ShellExecuteExW` which is called by `ShellExecuteA/W/ExA` on Windows) to significantly reduce the number of monitored syscalls. We also have to consider the frequency at which specific syscalls are triggered. To give a concrete example, let us assume we monitor the syscall `NtCreateFile`. When executing Office applications we might not experience a large number of new files, however, executing a web browser presumably increases the quantity of invocations considerably because of web content being cached. Furthermore, we have to select the syscalls based on the information they provide. Monitoring syscalls that solely pass handles or similar memory addresses might not be that effective since most of these addresses differ on each end host. Yet, intercepting syscalls operating on executable memory might lead to malicious activities on a client system. To comply with these restrictions, our approach mainly focuses on API and system calls that can be compared to blacklists, signatures, and reports gathered from automated malware analysis systems such as Anubis [7]. More precisely, we monitor syscalls providing information such as mutex-, file-, and service names, file hashes of executables, and IP addresses, domain names, URLs, and network messages to trace most of the outgoing communication to other end hosts such as botnet Command & Control (C&C) servers or SMTP servers for spam delivery.

3 Implementation

In this section, we introduce the caching mechanism utilized in our approach and describe the Cloud-to-client communication protocol. We then focus on the prototype of the Cloud service and the individual end host implementations.

3.1 Caching

When limiting the set of syscalls to those providing the information mentioned above, we would still have to handle a vast number of invocations by the client processes. To achieve a sufficient performance, our approach thus has to adopt an efficient caching strategy. As a result, the Cloud and the end host prototypes implement fast and cost-efficient Bloomfilter [8] caches to store and query already processed syscall arguments. Each prototype allocates three caching instances for every type of argument (e.g., filename and URL). Two instances store *trusted* ($\mathcal{T}$) and *malicious* ($\mathcal{M}$) entries, which are gathered from external sources. The third cache covers entries which are neither *trusted* nor *malicious* but were analyzed by the Cloud before. We name the last category *unsuspicious* ($\mathcal{U}$).

3.2 Communication

The communication between the Cloud and the clients is performed by interchanging a custom protocol that keeps the required network usage at a low level.

Notation: A syscall is denoted by its name and one or multiple arguments, defined as $\mathcal{S} := \{\ name,\ \mathcal{A}^+\ \}$. A syscall argument is represented by $\mathcal{A} := \{\ type,\ data,\ \mathcal{L}\ \}$. The parameter *type* denotes the argument type (e.g., filename or URL), and *data* contains the actual value of the argument. We define the label $\mathcal{L} := \mathcal{M} \mid \mathcal{T} \mid \mathcal{U} \mid \mathcal{TA} \mid \mathcal{ACP} \mid \mathcal{CR}$, whereas we distinguish between the categories *malicious*, *trusted*, *unsuspicious*, *temporarily approved*, *approved but caching prohibited*, and *caching revoked.*

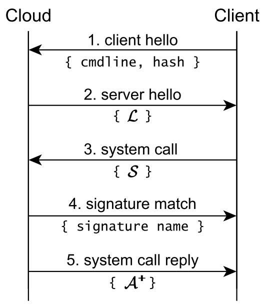

Fig. 1. Protocol

Protocol: As shown in Figure 1, the protocol mainly utilizes five distinct message types. The message `client hello` is sent by each running and newly executed client process and includes its command line and the file hash of the corresponding executable. To complete the two-way hand-shake, the Cloud service looks up the command line and file hash in its caches and transmits the `server hello` message including the analysis result $\mathcal{L}$. When the process is associated to an already known malicious executable, $\mathcal{L}$ is defined as *malicious* and security measures are applied, defined by the specified security policies. If the file hash is not cached at the Cloud, the hash value is requested at external data sources and analysis modules. Depending on the security policies, the execution of the syscall is either denied or delayed to prioritize the safety of the end host or temporarily approved by setting the label to $\mathcal{TA}$ to avoid a delay on the client. Once we receive the analysis results from the external modules, we transmit an updated `server hello` message including the final label to the client. Update messages, however, might not be received by a device, e.g., during offline phases, hence the host could unknowingly perform malicious activities. We thus implemented fail-over solutions in the individual end host prototypes.

The remaining three messages are exchanged when end hosts forward invoked syscalls to the Cloud service as discussed in the following.

3.3 Cloud Implementation

The prototype of our Cloud service is implemented as a light-weight, extensible Python script and leverages an external database containing data from malware analysis systems and blacklists. We also utilize the third-party services VirusTotal [9] and Google Safebrowsing [10] to obtain details about syscall arguments.

Figure 2 illustrates the processing stages of the Cloud service once it receives a `system call` message from a client process (1). First, the syscall $\mathcal{S}$ is compared to locally stored signatures, characterizing malicious behavior and security policies. As signatures may consist of multiple consecutively executed syscalls, we verify if the currently processed syscall is part of a signature and whether additional syscalls are required to match the complete signature. When a complete signature is triggered, we forward the `signature match` message including the name of the triggered signature to the client, invoke security measures, and skip further analyses (2). If no signature is triggered but the syscall was part of a signature, the labels of all arguments in $\mathcal{S}$ are set to *approved but caching prohibited* ($\mathcal{ACP}$)—unless they are flagged as *malicious* in the succeeding analysis steps. When prohibiting caching of these arguments, we require the end hosts to always forward the corresponding syscalls to the Cloud for repeated analysis. In step (3), the syscall arguments are extracted from $\mathcal{S}$ and then individually checked against the caches (4). When an argument is not cached, the database is queried (5). If the query was successful, the result is added to the appropriate cache and written to the label $\mathcal{L}$ of the argument $\mathcal{A}$ (6). If the argument is neither cached nor stored in the database, we forward the argument (i.e., file hash, URL, domain, or IP address) to external analysis modules (7) and continue processing the next arguments. Once an analysis result from an external module is returned to the Cloud, it is added to the corresponding cache (8). When all arguments of $\mathcal{S}$ are analyzed locally and external modules still process arguments, we either decline or delay the execution of the syscall to ensure the end hosts' security or temporarily approve the syscall (9), similar to the `server hello` message.

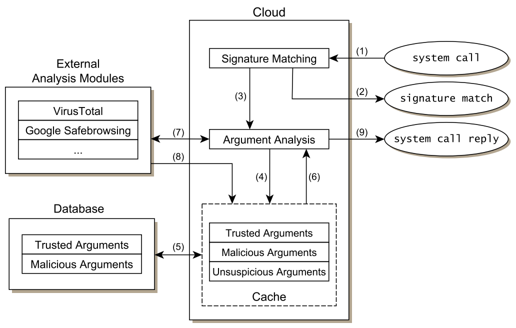

Fig. 2. Cloud implementation

On signature updates, we distribute *caching revoked* messages for all syscall arguments in the new signatures. This ensures that these arguments are removed from the client caches and always forwarded to the Cloud for signature matching.

3.4 Windows Implementation

Our prototype for desktop computers running Windows is split into two components, a background service and a syscall hooking library. The service is a light-weight application running in the background of each client system and utilizes the madCodeHook framework [11] to inject the library into each running and every new process. The hooking library is based on a heavily modified version of the `cuckoomon` library utilized by the Cuckoo Sandbox [12] and allocates a dedicated *hook function* for each monitored syscall. Once a syscall is invoked by a client process, the execution flow is redirected to the respective hook function, which first performs a look up of the syscall arguments in the locally stored caches. If not cached at the client, a `system call` message is sent to the Cloud. Depending on the results obtained from the caches or the Cloud, the execution of the native syscall function is then performed or prohibited.

As stated in Section 2, we have to closely select the monitored syscalls to achieve sufficient performance, thus we limit our monitoring to 29 syscalls and explain in the following the process how these system calls were chosen. To discover malware copying or renaming files to hide its presence, we monitor `NtCreateFile` and `NtFileOpen`. Hooking these syscalls, however, might induce a huge number of invocations, thus we limit the monitoring to two situations. We monitor `NtCreateFile` to obtain the filenames of newly created files. Note that we cannot perform any other investigations as no content is written yet. We also monitor both syscalls when the file contains a *Portable Executable* (PE) header, indicating a Windows executable. As the file paths provided by these syscalls might include client data such as the user name, we have to pre-process these paths and normalize user data with predefined values before forwarding the arguments to the Cloud to enable a comparison across multiple clients. We further hook syscalls responsible for DNS requests and opening URLs and monitor the socket functions `connect`, `send`, and `sendto` as these syscalls allow us to closely monitor target IPs and messages sent over the network. To only inspect the header data of a transmission, we limit the size of monitored messages to a minimum of 64 bytes and a maximum of 25% of the message length and only investigate the very first message sent over each socket. To protect the data privacy, we operate on hashed values only, thus split the network messages at specific delimiters, perform cryptographic hash operations on each argument individually, and look up every hash value in a cache covering malicious message fragments (e.g., keywords used by malware). As malware often creates distinct mutex names when probing for an already infected client system and installs itself as a service using a specific name, we also monitor syscalls related to these events. To detect and prevent the execution of malware at the earliest possible time, we trace `ShellExecuteExW` and `CreateProcessInternalW`.

To also track malicious activities in offline phases, we implement multiple failover solutions. First, we rely on adjustable security policies such as terminating the application or prohibiting specific types of syscalls (e.g., network operations) once uncached syscalls are invoked. Further, the hooking library maintains a local storage in which invoked syscalls are logged, while we approve or decline the syscalls depending on the security policies. Once the connection to the Cloud is restored, the recorded syscall information is replayed. The end host further logs invoked syscalls once a syscall is temporarily approved until an updated analysis result is received. To prevent manipulations by malware, we make use of *secure log files* [13] in which each entry is part of a cryptographic hash chain to validate all previous entries. We acknowledge that this solution does not protect the device against getting infected with malware, yet, the Cloud will block an infected device once the syscalls resembling the malicious behavior are replayed.

3.5 Android Implementation

To also evaluate our approach on mobile devices, we developed a prototype for Android, a middleware running a modified Linux kernel. We split our prototype into two components, a kernel module and a Java application. The kernel module is the sensitive part of our implementation since even minor issues can destabilize the entire OS. We thus implemented the syscall hooking in the kernel and moved less essential components such as caching and the Cloud communication into the app. Similar to Windows, we implemented hook functions to intercept syscall invocations. Again, these syscalls were systematically chosen to have only limited impact while maintaining a good visibility into the behavior of the system. To monitor file operations, we trace seven file syscalls (e.g., `sys_creat` and `sys_rename`). We again monitor the network operations `sys_connect`, `sys_send`, and `sys_sendto` and shorten the inspected messages. Equally to the Windows prototype, we also track the execution of new processes, defined by `sys_execve`. In order to filter syscalls of presumably benign default Android processes, we implemented a whitelist containing paths of common processes, files, and IP addresses used for inter-process communication. After whitelisting carefully-chosen services and filenames such as *SensorService* and */dev/urandom*, we were able to reduce the noise of syscall invocations considerably.

When a process invokes a monitored syscall, the name and path of the process and carefully-selected syscall arguments are checked against the whitelist. If whitelisted, the syscall is approved and the process execution is resumed immediately, otherwise the syscall is forwarded to the app, which checks if the syscall arguments are stored in the local caching instances. If not cached, we forward the syscall to the Cloud. Depending on the results obtained from the caches or the Cloud, the execution of the native syscall function is performed or prohibited.

Particularly for mobile devices, we cannot rely on a stable network connectivity and have to provide fail-over solutions during offline phases (e.g., loss of signal). Similarly to the Windows prototype, the app maintains a local storage in which invoked syscalls are logged using secure log files, while we approve or decline the syscalls depending on the security policies. Again, this information is replayed once the connection to the Cloud service is restored.

3.6 Signature Generation

There already exists a large body of work on signature generation based on syscalls [1, 14, 15], thus we do not focus on that part and stick to a straightforward way to generate the signatures for the evaluation of our approach. Note that arbitrary signature generation algorithms can be used. To generate signatures based on invoked syscalls, we execute samples of various malware families in a virtualized analysis environment and record the invoked syscalls. We then search for sequences of syscalls that can be found in a certain amount of the samples using a *longest common substring* (*LCS*) algorithm. If a LCS of syscalls is found, it is known to be characteristic for the specific malware family. We then apply the *Levenshtein* distance function to measure the similarity of these

LCS sequences and their arguments. Matching on syscall arguments, however, can be problematic as specific types of values are defined by a certain amount of randomness, e.g., file-, or mutex names. To compensate randomness, we first compute the LCS using the syscall names only (e.g., `NtCreateFile`), ignoring any arguments. Afterwards, we combine the syscall names with the set of arguments that have been observed. To perform signature matching, we again apply the distance function to detect similarities between signatures and invoked syscalls. We verified our signatures against benign sample sets as discussed in the next section to avoid side effects caused by common system operations.

4 Evaluation

We now discuss the results of the performed experiments to verify the applicability and reliability of our approach. The evaluation of our Windows prototype is conducted on a desktop computer using an Intel i7-2600 CPU with 3 GB of memory and Windows XP (SP 3)—later versions of Windows, however, can also be deployed. We chose Windows XP to leverage the analysis reports of various automated malware analysis systems such as Anubis, which are mostly still running Windows XP. The mobile prototype is evaluated on a *Samsung Galaxy Nexus* device using Android 4.1.2 and kernel version 3.0.31, utilizing WiFi to connect to the Cloud service. The prototype further supports the official Android emulator using Android 4.2 and kernel 2.6.29 to perform automated analyses of Android malware samples. During our evaluation, the network latency between the desktop computer and the Cloud, directly connected via LAN, resulted in an average of 0.52 ms, respectively, 11.7 ms between mobile device and the Cloud.

4.1 Caching

We first evaluate the influence of caching on the performance of the individual end host implementations. While conducting this experiment, we disabled external modules and signature matching to only measure the impact of our caching implementation and the Cloud-to-client communication. On the desktop computer, we perform the following experiment consisting of four different tests: i) we create empty files using unique filenames, ii) create a substantial number of unique mutexes, iii) repeatedly execute a dummy application that terminates right after invocation, and iv) allocate multiple sockets and establish connections to an external host in the local network without transmitting data. We repeat each test ten times and calculate the average on the measured execution times. To determine the impact of caching, we

Table 1. Performance impact of caching

	Native	Disabled		Cloud		Full	
Test	sec.	sec.	%	sec.	%	sec.	%
Windows:							
1000 Files	2.74	3.57	130	3.04	111	2.78	101
2000 Mutexes	0.03	1.17	3,900	0.82	2,733	0.03	100
100 Processes	0.52	0.69	133	0.65	125	0.64	123
1000 Sockets	0.31	0.79	255	0.68	219	0.31	100
Android:							
200 Files	0.20	1.89	945	1.76	880	0.37	185
100 Processes	2.55	5.71	224	5.62	220	2.59	102
200 Sockets	1.06	2.33	220	1.98	187	1.20	113

run this experiment four times: i) without injecting our hooking library (native Windows system), ii) injected library but disabled caches, iii) enabled caching at the Cloud, iv) enabled caching at the Cloud and the client side. We perform almost the same experiment on the Galaxy Nexus, except that we skip Test 2 as Android does not support mutexes. Before starting a test, we flush the caches to make sure each test run has the same preconditions.

As outlined in Table 1, each individual test lasts considerably longer when caching is disabled. Enabling the Cloud caches reduces the number of requests sent to the database as we can rely on already fetched results. Activating the caches on the client has the largest influence on the execution time since the client does not interrupt the execution to wait for analysis results from the Cloud. In fact, full caching improves the performance to a level where the execution times almost approximate to the results when no syscalls are monitored.

4.2 Windows

Caching has a huge impact on the performance of our Cloud-based approach, thus we attempt to cache as many syscall arguments as possible by creating a ground truth of trusted arguments, which can be used as initial values for the caching instances. To obtain a ground truth set, we set up a fresh installation of Windows, installed commonly used software such as browsers, file archivers, and Office software and executed each application for a few minutes.

Table 2. Cached (C) and uncached (U) syscalls invoked by Windows software

Software	C	U	Total
Adobe Reader	27	3	30
Google Chrome	98	10	108
Internet Explorer	62	5	67
Mozilla Firefox	13	0	13
Ms Media Player	15	1	16
Ms Paint	9	1	10
Notepad	8	0	8
Regedit	8	0	8
Services.msc	197	0	197
Skype	43	2	45
Taskmanager	2	0	2
WinZip	26	1	27
Total	508	23	531

Software: To determine the number of syscalls that still have to be forwarded to the Cloud when full caching is enabled, we enumerate the cached and uncached syscalls of common software, as depicted in Table 2. The software was already installed when generating the ground truth, thus each program was executed at least once. We find 508 syscalls cached at the client and 23 syscalls not stored in the caches. Without caching, we would have to forward 531 syscalls to the Cloud, thus caching reduces the communication between Cloud and clients significantly. Executing unknown software certainly requires a higher number of requests to the Cloud on the first execution, however, on second execution, most of these syscalls presumably are also cached.

Signatures: To evaluate the feasibility of a Cloud-based signature matching, we leverage the set S_{Mal} covering 1,508 clustered malware samples [16]. This set includes 13 malware families. To obtain signatures, we apply the algorithm introduced in Section 3.6 on all samples of each family using a Levenshtein-ratio of 90%. The signature length (i.e., the number of consecutive syscalls required by a signature) varies between one and nine syscalls.

To verify the correctness and detection capabilities of the signatures, we analyzed 234,829 samples randomly taken from the malware analysis service Anubis since August 2012 and obtained 18,493,498 syscalls to perform signature matching upon. When comparing our signatures against these syscalls, we find 3,323 signature matches. To verify that all matches hit a malicious executable, we request analysis results of various malware detection services from VirusTotal. As stated in Table 3, we find 3,406 Anubis samples to belong to one of the malware families in S_{Mal}, while the remaining samples are associated to other families not covered by our signature generation set. 2,140 matches (62.8%) are detected as the exact malware variant as found in S_{Mal} and are therefore considered correctly identified by our signatures. For 1,027 samples (30.2%), the signatures do not hit the exact variant of the family as stated in S_{Mal}, yet the VirusTotal results imply that we detected a different variant from the same family. This shows that our approach has the capability to tolerate differences malware authors presumably integrate to evade detection by antivirus software. For 156 signature matches (4.6%), VirusTotal results include different family names than stated by our signatures. When manually checking the samples, we discovered multiple *Sality* samples to be erroneously flagged by the antivirus vendors. Overall, our signatures correctly identified 97.6% of the malware samples as malicious. The false negative rate (i.e., the samples that are not detected by our signatures) is at 2.4%, which is mostly caused by the family *Spygames*. The signature is a single `send` call that transmits a partially randomized string. To detect this family we would have to set the *Levenshtein* ratio to a value below 50%, however, that would lead to thousands of false positives for the other signatures. Other malware families in the set S_{Mal} (e.g., *Adultbrowser*) are not included in the Anubis sample set at all, thus we cannot evaluate our signatures for these families.

Table 3. Signature matching results (EF = Exact Family, FV = Family Variant, FM = Family Mismatch, FN = False Negative)

		(in %)			
Family	#Samples	EF	FV	FM	FN
Allaple	1,951	99.3	0.6	0.0	0.1
Bancos	33	9.1	36.4	54.5	0.0
Casino	27	0.0	100.0	0.0	0.0
Flystudio	38	0.0	10.5	89.5	0.0
Magiccasino	1	100.0	0.0	0.0	0.0
Poison	54	50.0	20.4	14.8	14.8
Porndialer	3	0.0	0.0	0.0	100.0
Sality	1,239	13.9	77.6	7.7	0.8
Spygames	60	0.0	0.0	0.0	100.0
Total	3,406	62.8	30.2	4.6	2.4

Samples submitted to a malware analysis system commonly are of malicious character. Yet, according to VirusTotal the Anubis data set also contains 14,729 unsuspicious samples of which none is falsely classified to be malicious by our signatures. To further verify that our signatures are not triggered by benign software, e.g., ordinary Windows software, we perform three additional experiments: i) we prepare a system with five web browsers (i.e., Mozilla Firefox, Internet Explorer, Google Chrome, Opera, and Apple Safari), disable Flash, Java, and JavaScript and visit the Alexa Top 5,000 websites [17] twice with each browser, ii) repeat the experiment in i), whereas Flash, Java, and JavaScript are enabled, and iii) set up a fresh Windows host and manually install and execute several types of updates, commonly used software, and games. In total, 1,058 different applications are executed. While performing these experiments, we compared the invoked syscalls against our malware signatures. Overall, 81,256,875 syscalls with 624,125,933 individual arguments are invoked. As aimed for, we do not experience a single hit of a signature, thus assume these signatures to be reliable to protect against known malware without classifying benign software to be malicious.

The third experiment simultaneously serves as a survey to evaluate the user experience, i.e., whether noticeable delays or problems are encountered. We repeat this test three times and find none of the 1,058 applications causing any problems such as crashes or error messages. For the majority of applications, we do not encounter any noticeable delays, however, while installing specific software (e.g., Microsoft Visual Studio 2012), which copies thousands of files onto the hard disk drive, we observe minor delays during the installation phase of the first run. These delays emerge as most of the accessed filenames are neither included in the initial ground truth nor cached at the Cloud service. When performing the second and third run, the filenames are still not included in the initial ground truth of the client (which is wiped after each run), but stored in the Cloud caches, resulting in almost no noticeable delays during these runs.

4.3 Android

We again build a ground truth data set containing syscall arguments of pre-installed and therefore likely benign software. Similarly to Windows, we execute these apps and classify each invoked syscall argument as trusted.

Table 4. Cached (C) and uncached (U) syscalls invoked by pre-installed apps

Application	C	U	Total
Browser	25	2	27
Calculator	25	1	26
Calendar	19	1	20
Contacts	20	1	21
Deskclock	20	1	21
Gallery	26	2	28
MMS	25	0	25
Settings	21	1	22
Videoeditor	23	1	24
Total	204	10	214

Apps: We execute nine pre-installed apps to determine how many syscalls are not covered by the ground truth and have to be analyzed by the Cloud. As shown in Table 4, we experience a small amount of uncached syscalls as most of the data is already stored in the caches.

Android is heavily built around the feature to install third-party apps, thus we also investigate how many syscalls are uncached when executing external apps for the first, respectively, second time as their syscall arguments might not be covered by the ground truth. As depicted in Table 5, we select ten commonly used apps and record the number of syscall invocations. We execute each app once to remove potential welcome screens and specify credentials of test accounts for specific apps (e.g., *Facebook* and *Twitter*). We then reset the caches to the initial ground truth values to ensure that none of the values got cached when preparing these apps. When executing the apps for the first time, 516 syscalls are already covered by the ground truth, yet we still have to analyze 221 unknown syscalls at the Cloud. When executing the apps a second time, the caching strategy requires merely 25 syscall analyses at all, most of them caused by *Instagram*. We thus argue that caching improves the performance of our approach significantly, on desktop computers and mobile devices for known and unknown software.

Signatures: To evaluate the Cloud-based signature matching on Android, we select two distinct malware families, namely the spy app *Gone in 60 seconds* (*Gi60s*) and the banking trojan *Carberp*. We execute one sample of each family and manually extract a signature based on the invoked syscalls. The signature of *Gi60s* depends on two consecutive `sys_sendto` syscalls that forward user data to external servers. The signature of *Carberp* is based on five syscalls invoked at the initialization phase of the app. To validate the signature matching we execute four different samples of *Gi60s* and two samples of *Carberp* on the mobile device and find our signatures to correctly classify all samples as malicious.

5 Discussion

The main argument to move the detection of malicious activities from end hosts to a Cloud service is that we assume the end hosts to be significantly less secure compared to the Cloud since a centralized system is maintained by experienced personal. Especially the currentness of detection algorithms and signatures is presumably better on a Cloud-based service than on individual end hosts, mostly maintained by regular users. Since operators have to maintain a single service only, updates of the detection mechanisms can be conducted in an easy way to quickly react to incidents or to enforce security policies. Further, the centralized strategy permits the operators to closely monitor and quarantine recently infected clients. If operators would have to be responsible for updating the security software on every client individually, they probably would be overwhelmed by the number of clients, especially when also considering mobile devices that are becoming more popular.

Table 5. Cached (C) and uncached (U) syscalls invoked by commonly used apps

	1st run			2nd run		
Application	C	U	Total	C	U	Total
Adobe Reader	11	3	14	14	0	14
Angry Birds	36	19	55	50	0	50
Google Chrome	46	71	117	108	3	111
Facebook	191	12	203	101	0	101
Mozilla Firefox	53	38	91	83	0	83
Instagram	31	18	49	33	18	51
Shazam	35	14	49	47	2	49
Twitter	47	7	54	38	0	38
VLC Player	4	4	8	27	1	28
Winamp	62	35	97	93	1	94
Total	516	221	737	594	25	619

The Cloud service is connected to existing firewall and intrusion detection systems to immediately apply security measures such as blocking a client from accessing particular infrastructure in the case of an incident. Further, when a client connects to the network, it is limited to communicate with the Cloud service at first. If the client does not instantiate a connection to the Cloud or stops communicating with the Cloud (e.g., after approving a syscall temporarily), the device is flagged as potentially dangerous and its network connectivity is revoked. A client not communicating with the Cloud can have multiple reasons: either the host is not participating in the Cloud-based security strategy yet (e.g., a new mobile device) or the device got infected during an offline phase and malware deactivated the security mechanisms. Nevertheless, as long as unprotected and infected clients are blocked from the network, no other clients can be harmed.

Applying a Cloud-based security approach, however, might also raise severe drawbacks. Forwarding syscalls to the Cloud and delaying the execution of end hosts' processes causes an overhead due to the high number of invoked syscalls on every client. Without selecting a partial set of syscalls to monitor and applying efficient caching, this would be a serious issue for the clients' performance and users' experience. Our approach thus makes exhaustively use of mechanisms to limit the number of requests to the Cloud. The evaluation results indicate that the overhead is reasonable in practice as almost no noticeable delays are induced.

A further limitation of a Cloud-based approach is the requirement of a continuous network connectivity, which cannot be guaranteed for specific types of end hosts like mobile devices. Still, we consider our approach to be feasible in well-established networks in which most of the clients such as desktop computers are permanently connected. Mobile devices could fall back to UMTS/GSM in areas without WiFi connectivity, however, GSM networks are significantly slower than WiFi connections. When evaluating the impact of caching in Section 4.1, creating 200 files or executing 100 processes both required 8 seconds using GSM and full caching. Yet, the number of uncached syscalls highly depends on the app, as shown in Table 5. Many apps invoke only a limited number of uncached syscalls that can also be transferred in a reasonable time using GSM networks.

A general limitation of a network service is the central point of failure. This problem can be avoided by providing fail over solutions to ensure the availability of the centralized system. Further, we can split the Cloud service into one master node and multiple slaves to distribute the load and to make the service resilient to faults of single servers. An additional benefit is that clients connect to the nearest node to reduce the network latency and the delay on the clients' side.

A centralized protection service is a promising target for adversaries, e.g., by infiltrating or taking down the service. The Cloud service thus has to be protected by reasonable security measures and monitored closely to identify and prevent attacks. It also has to be taken care of end hosts attempting to trick the Cloud, e.g., by malware taking over the clients' security application and emulating the communication to the Cloud. A solution includes the usage of a kernel-based client application. Encrypted and signed communication channels between Cloud and clients are mandatory to protect the integrity and confidentiality of the inter-communication. Secure channels also eliminate the risks of eavesdropping, replay, man-in-the-middle, and other serious attacks. We further have to address the issue of processing plain syscall arguments at a network service as proprietary data is relayed to a centralized system not under the control of the individual user. Yet, this drawback can be bypassed, e.g., by operating on hashed arguments only. Information sent to the Cloud thus cannot be converted back to plain text, hence we gain a privacy preserving approach, assuring the users' confidentiality. The functionality of detection mechanisms is not affected by hashed arguments, yet techniques such as blacklist comparison need to be altered to operate on hash values. When switching to hash functions, we have to ensure the hashes to be resilient against attacks, e.g., by applying *Hash-based Message Authentication Codes* [18] using individually shared keys between the Cloud and the end hosts.

6 Related Work

Analyzing system calls to detect malicious behavior has a long history on desktop computers. Some approaches [1, 19] develop benign behavior profiles based on multiple consecutively invoked system calls to identify anomalous behavior. Mutz *et al.* [20] analyze the relationship between system call arguments and the invocation context to detect malicious actions. Stinson and Mitchell [2] perform botnet detection based on system calls in combination with tainting untrusted memory values. Srivastava and Giffin [21] propose an approach that combines the analysis of network traffic with a hypervisor-based identification of malicious behavior at the user-, and kernel-level. Burguerae *et al.* [22] show that system call analysis is also feasible on mobile devices.

The idea of detecting malicious software at a centralized service is already explored in many approaches [3–5]. Oberheide *et al.* [3] present CloudAV, which utilizes a light-weight application running on end hosts to suspend the execution of an unknown binary, forward the binary to the Cloud, and perform or decline its execution based on the analysis result of the Cloud. Furthermore, Oberheide *et al.* [5] discuss an approach to move CloudAV from desktop computers to mobile devices. A more sophisticated approach is presented by Martignoni *et al.* [23] in which users may delegate the execution of potentially malicious applications to a Cloud service. As a result, the unknown process is executed in the Cloud, however, by interchanging specific system calls, the application acts like it is executed locally on the client.

7 Conclusion

We introduced a Cloud-based security approach to move the detection of malicious activities from individual end hosts to a centralized network service. To determine if activities on client systems are malicious, every end host forwards selected API and system calls to a Cloud service and awaits approval or denial to execute these operations locally. To evaluate the applicability of our approach, we implemented prototypes for desktop computers and mobile devices and found this protection strategy to be feasible in practice as almost no delays are caused on the client which would interfere with the usability of the end hosts.

Acknowledgment: This work was supported by the German Federal Ministry of Education and Research (Grants 16BY1110/MoBE and 16BY1020/MobWorm).

References

1. Forrest, S., Hofmeyr, S., Somayaji, A.: The Evolution of System-Call Monitoring. In: Proceedings of the 2008 Annual Computer Security Applications Conference. ACSAC '08, Washington, DC, USA, IEEE Computer Society (2008) 418–430
2. Stinson, E., Mitchell, J.C.: Characterizing Bots' Remote Control Behavior. In: Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA '07, Berlin, Heidelberg, Springer-Verlag (2007) 89–108
3. Oberheide, J., Cooke, E., Jahanian, F.: CloudAV: N-Version Antivirus in the Network Cloud. In: Proceedings of the 17th Conference on Security Symposium. SS'08, Berkeley, CA, USA, USENIX Association (2008) 91–106
4. Harrison, K., Bordbar, B., Ali, S.T.T., Dalton, C.I., Norman, A.: A Framework for Detecting Malware in Cloud by Identifying Symptoms. In: Proceedings of the 2012 IEEE 16th International Enterprise Distributed Object Computing Conference. EDOC '12, Washington, DC, USA, IEEE Computer Society (2012) 164–172

5. Oberheide, J., Veeraraghavan, K., Cooke, E., Flinn, J., Jahanian, F.: Virtualized In-Cloud Security Services for Mobile Devices. In: Proceedings of the First Workshop on Virtualization in Mobile Computing. MobiVirt '08, New York, NY, USA, ACM (2008) 31–35
6. Gu, G., Perdisci, R., Zhang, J., Lee, W.: BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. In: Proceedings of the 17th Conference on Security Symposium. SS'08, Berkeley, CA, USA, USENIX Association (2008) 139–154
7. Bayer, U., Krügel, C., Kirda, E.: TTAnalyze: A Tool for Analyzing Malware. In: Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference. (4 2006)
8. Knuth, D.E.: The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998)
9. Virustotal: VirusTotal Private API v2.0 (2014)
10. Google: Safe Browsing API v2.0 (2014)
11. Rauen, Mathias: madcodehook Framework (2014) `http://madshi.net/`.
12. Guarnieri, Claudio: Cuckoo Sandbox (2014) `http://www.cuckoosandbox.org/`.
13. Schneier, B., Kelsey, J.: Secure Audit Logs to Support Computer Forensics. ACM Trans. Inf. Syst. Secur. **2**(2) (May 1999) 159–176
14. Wang, L., Li, Z., Chen, Y., Fu, Z., Li, X.: Thwarting Zero-Day Polymorphic Worms With Network-Level Length-Based Signature Generation. IEEE/ACM Trans. Netw. **18**(1) (February 2010) 53–66
15. Wurzinger, P., Bilge, L., Holz, T., Goebel, J., Kruegel, C., Kirda, E.: Automatically Generating Models for Botnet Detection. In: Proceedings of the 14th European Conference on Research in Computer Security. ESORICS'09, Berlin, Heidelberg, Springer-Verlag (2009) 232–249
16. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic Analysis of Malware Behavior using Machine Learning. J. Comput. Secur. **19**(4) (2011) 639–668
17. Alexa Internet, Inc.: Top 1,000,000 Websites (2014)
18. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '96, London, UK, UK, Springer-Verlag (1996) 1–15
19. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion Detection using Sequences of System Calls. J. Comput. Secur. **6**(3) (August 1998) 151–180
20. Mutz, D., Robertson, W., Vigna, G., Kemmerer, R.: Exploiting Execution Context for the Detection of Anomalous System Calls. In: Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection. RAID'07, Berlin, Heidelberg, Springer-Verlag (2007) 1–20
21. Srivastava, A., Giffin, J.: Automatic Discovery of Parasitic Malware. In: Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection. RAID'10, Berlin, Heidelberg, Springer-Verlag (2010) 97–117
22. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: Behavior-Based Malware Detection System for Android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices. SPSM '11, New York, NY, USA, ACM (2011) 15–26
23. Martignoni, L., Paleari, R., Bruschi, D.: A Framework for Behavior-Based Malware Analysis in the Cloud. In: Proceedings of the 5th International Conference on Information Systems Security. ICISS '09, Berlin, Heidelberg, Springer-Verlag (2009) 178–192