PhantomFS: File-Based Deception Technology for Thwarting Malicious Users

JUNGHEE LEE1, (Member, IEEE), JIONE CHOI1, GYUHO LEE2, SHIN-WOO SHIM2, AND TAEKYU KIM2
1School of Cybersecurity, Korea University, Seoul 02841, South Korea
2Cyber Warfare Research and Development Laboratory, LIG NEX1 Company, Ltd., Seongnam 13488, South Korea
Corresponding author: Junghee Lee (j_lee@korea.ac.kr)

ABSTRACT File-based deception technologies can be used as an additional security barrier when adversaries have successfully gained access to a host evading intrusion detection systems. Adversaries are detected if they access fake files. Though previous works have mainly focused on using user data files as decoys, this concept can be applied to system files. If so, it is expected to be effective in detecting malicious users because it is very difficult to commit an attack without accessing a single system file. However, it may suffer from excessive false alarms by legitimate system services such as file indexing and searching. Legitimate users may also access fake files by mistake. This paper addresses this issue by introducing a hidden interface. Legitimate users and applications access files through the hidden interface which does not show fake files. The hidden interface can also be utilized to hide sensitive files by hiding them from the regular interface. By experiments, we demonstrate the proposed technique incurs negligible performance overhead, and it is an effective countermeasure to various attack scenarios and practical in that it does not generate false alarms for legitimate applications and users.

INDEX TERMS Deception technology, file system, honeypot.

I. INTRODUCTION
Deception technology (often called honeypot) is an information security technique that detects, deflects or counteracts cyberattacks by using a fake system or data [1]. Compared to traditional perimeter or signature-based intrusion detection and anomaly detection techniques, it is known to be more effective in insider threats, social engineering, and 0-day attacks [2].

File-based deception technology can be employed to thwart malicious users when they have successfully gained access to a host. The file-based deception technology plays as an additional security barrier when intrusion detection/prevention systems fail to detect/prevent malicious users. It can be used to detect the existence of malicious users and to prevent them from accessing sensitive files. Existing file-based deception technologies have been applied only to user data files. If malicious users or applications access the fake user data files, it is considered as a symptom of intrusion. If we extend this concept to system files, we expect it will provide a strong countermeasure to various types of attacks. This is because it is very difficult (if not impossible) to commit an attack without accessing a single file. Though there are transient attacks which do not make any changes to files, it does not mean they do not read any file. In order for attacks to be successful, it is essential to gather information about the victim system. The information is usually gathered by reading system files. Without accessing system files, the accessible information must be very limited.

Though it is expected to be very powerful, there is a big obstacle to utilize fake files to detect malicious users: false alarm. In general, deception technologies are expected to offer a low false alarm rate [2]. However, fake files may generate excessive false alarms [3]. Legitimate users may access fake files by mistake. More seriously, the legitimate system services such as file indexing or searching frequently scan all files in the system, which generates false alarms by accessing fake files.

To the best of our knowledge, there is no known solution to this issue. In this paper, we address this issue by introducing hidden interface. The hidden interface is a system call that is used to access real files excluding fake files. Through the
hidden interface, fake files are not shown. Thus, legitimate users and system services, which use the hidden interface, cannot trigger false alarms. Through the regular interface, which is a set of the traditional system calls used to access files, fake files are shown. If any fake file is accessed through the regular interface, it will be reported to the administrator as a potential attack. Malicious users may want to steal sensitive files. We can also utilize the hidden interface to hide sensitive files. Sensitive files are allowed to be accessed only through the hidden interface while they are not even shown through the regular interface. In addition to these, there could be more use cases, which are discussed in Section IV.

In this paper, we propose PhantomFS that implements the file-based deception technology through the hidden interface. We implemented it by modifying the file system of Linux. Experimental results show that it incurs negligible performance overhead in reading and writing data files, and reading a directory compared to the unmodified file system. It is also demonstrated that PhantomFS is effective in detecting malicious users and preventing them from accessing sensitive files.

The rest of this paper is organized as follows. We discuss related works in Section II. After presenting an overview of PhantomFS and its use cases in Sections III and IV, we explain details of its implementation in Section V. Experimental results are given in Section VI followed by conclusions in Section VII.

II. RELATED WORKS

A. DECEPTION TECHNOLOGY

The deception technology has been applied to various entities of systems. Network-based deception technologies deflect cyberattacks by showing fake network entities such as fake network configurations, servers, services, and messages [4]. They make it hard for adversaries to find the real target on the network perimeter. To deceive adversaries who have gained access to a host, host-based deception technologies have been studied. Fake database [5], [6], password [7], account [5], and patch [8] are used to allure adversaries [2].

Using fake files is one of the host-based deception technologies. If an adversary accesses a fake file, it is reported as a potential attack [3], [9]–[11]. One of challenges the previous works try to address is how to make fake files as real as possible. To address this challenge, HoneyGen is proposed, which automatically generates fake files by profiling existing real files [12]. Fake files are also utilized to detect ransomware [13]. These techniques are effective in enhancing the effectiveness of alluring adversaries, but we have not found any previous works on addressing false alarms generated by fake files.

Previous file-based deception technologies offer simulation only (showing fake files), whereas PhantomFS offers not only simulation, but also dissimulation (hiding real files) and combination of simulation and dissimulation. By using these capabilities, PhantomFS maximizes the chance of successful deceit. For example, combination of simulation and dissimulation enables the content of a fake file to be identical with a real file, which make it very hard for adversaries to distinguish them.

B. SYSTEM CALLS

PhantomFS implements the hidden interface by modifying the system calls. The purpose of modifying the system call is to provide the hidden interface to the application. There are previous studies on utilizing system calls for the purpose which is different from their original intention.

Lauren et al. propose to diversify library entry points to system calls [14], [15]. Since entry points are unique in each computer, it becomes very difficult for malware to access system resources. The goal of this technique is to make it hard to find the correct system call.

Fake honeypot was studied to scare attackers away by making a host look like a honeypot. The host is made appear as a honeypot by placing honeypot tools, mysterious processes and unknown system calls [16]. If attackers are convinced that the victim host is a honeypot, they are very likely to leave the host. Non-standard system calls are used to deceive attackers.

Honeypots are originally used to collect and analyze behavior of attackers. System calls are often monitored to observe their behavior. Recent approaches use virtual machine to monitor system calls [17]–[20].

C. DIFFERENT FILE VIEWS

The concept of showing different files to different users at the same host may seem similar with what is offered by virtual machines and containers. However, the virtual machines and containers are not intended for deception. If one of virtual machines is compromised, other virtual machines on the same host may not be affected, but the data in the compromised virtual machine may be leaked or destroyed. In addition, PhantomFS allows users to configure the access interface of files individually, while virtual machines and containers do not support such fine-grained configuration.

D. INTRUSION DETECTION SYSTEMS

Intrusion detection systems (IDSs) are categorized mainly into network-based IDSs (NIDSs) [21], [22] and host-based IDSs (HIDSs) [23]–[25]. NIDS monitors network packets while HIDS monitors activities on the host to detect malicious behaviors. They typically detect malicious behaviors by matching the signature of malicious activities [24] or finding abnormal behaviors deviated from the known normal profile [25]. The artificial intelligent technique has also been studied in IDSs [21]–[23].

The purpose of IDSs is typically to prevent malicious users from accessing hosts and hindering normal operations. However, there still exists the possibility of malicious users penetrating the host evading IDSs. In this case, PhantomFS may play as an additional security barrier for thwarting malicious users.
III. OVERVIEW OF PHANTOMFS

This section provides an overview of PhantomFS. We explain key ideas of PhantomFS in this section and presents details of their design and implementation in Section V.

As illustrated in Figure 1, PhantomFS provides a hidden interface in addition to the regular interface. The files shown through the hidden interface are real files that are used by legitimate users and applications. The files shown through the regular interface are different and under monitoring. PhantomFS allows the administrator to configure through which interface a file should be shown. The administrator can also configure what type of access (read or write) should be reported.

For PhantomFS, a file maintains 4 additional flags as summarized in Table 1. If flag $h$ is set, the file is hidden from the regular interface. It is typically used to hide a sensitive file. If flag $r$ is set, it is reported to the administrator if the file is read through the regular interface. Similarly, if flag $w$ is set, write access to the file is reported. If flag $f$ is set, the file is hidden from the hidden interface. It is typically used for a fake file to avoid false alarm. These flags are also applicable to directories. For example, if flag $r$ is set for a directory, it is reported if the list of files in the directory is retrieved. By using these 4 flags, PhantomFS supports various use cases, which are discussed in Section IV.

It should be noted that hiding files and reporting file access are orthogonal to access control. Even if flag $r$ or $w$ is set for a file, if the user or application does not have privilege to access the file, the file cannot be accessed. If flag $h$ is set, the file is not shown, but can be accessed if its path is known a priori.

The hidden interface can be implemented by introducing a new system call, or using an existing system call in a different way. In our implementation, we use the read system call to implement the hidden interface. Before an application calls the read system call, the application allocates a buffer to receive the data. Since the buffer is supposed to be overwritten by the system call, it does not have any meaningful data. For PhantomFS, a request data structure is written to the buffer before calling the read system call. When the read system call recognizes the request, it processes the request. Otherwise, the read system call works in a regular way. Details of hidden interface implementation are discussed in Section V.

The legitimate applications should use the hidden interface and the legitimate users should use the legitimate applications. Applications usually do not directly call system calls, but they call wrapper functions provided by the standard library. Thus, we provide a modified library that uses the hidden interface. Legitimate applications do not need to be re-compiled, but their executable needs to be modified by the binary editing tool to change their linked library, as illustrated in Figure 2.

The modified executables are placed in a hidden directory. Recall that flags are applicable to directories and hidden files/directories are still accessible if their path is known a priori. Therefore, legitimate users can use the modified executables in the hidden directory. For convenience, a script is provided to change the default search paths to the hidden directory. As long as the legitimate users do not forget to run the script, there is no reason for them to access fake files by accident.

For example, the ls utility usually locates in /bin directory. We provide its modified executable in /a_hidden_path with the same name. A script is provided to change the PATH environment variable to /a_hidden_path. /a_hidden_path is just an example, and an arbitrary name can be used as a hidden path. After running the script, the user can use ls as if nothing has changed. Therefore, the false alarm caused by mistakes of users can be drastically reduced.

The hidden path and the script are hidden i.e. their flag $h$ is set. They can be accessed only by users who know their correct path. Thus, the path name itself plays as the same role with a password. Since the hidden path is analogous to a password, legitimate users should remember it and it should not be exposed to unauthorized users.
TABLE 2. Use cases of PhantomFS with different flags.

| Flags | Use case                      |
|-------|------------------------------|
| (none)| Regular files/directories    |
| rwf   | Fake files/directories       |
| h     | Hidden files/directories     |
| w     | Tamper-resistant files/directories |
| rw    | Live decoy files/directories |

IV. USE CASES

This section presents use cases of PhantomFS. Before we present use cases, we define the threat model in the following subsection.

A. THREAT MODEL

PhantomFS targets at alluring adversaries who have gained access to a host evading intrusion detection systems. The adversaries can be local (those who access the computer through a local terminal) or remote (those who access the host over a network). The adversaries may run malicious applications (malware) or use existing applications or utilities for their purpose. They may obtain the administrative privilege by exploiting vulnerabilities of the host. The goal of PhantomFS is to detect their malicious behavior on the host by detecting access to decoy files through the regular interface and to prevent them from accessing sensitive files.

PhantomFS does not assume kernel-level malware (rootkits). Since it is implemented in the file system, if rootkit exists, its correct operation cannot be guaranteed.

In this paper, it is assumed that adversaries are not aware of PhantomFS. If they are, they will try to avoid detection mechanisms offered by PhantomFS. Indeed, it is a common issue of existing deception technologies. If adversaries are aware of those deception technologies (such as honeyfile, honeyword, honey sheets, etc.), they are unlikely to be deceived by those technologies. We will discuss intelligent adversaries who know and try to nullify PhantomFS in our future work.

Precisely, we assume that adversaries do not know the fact that PhantomFS is installed on the host they are accessing. In addition, they do not know the hidden path and the existence of the hidden interface. The legitimate applications and the modified library locate in the hidden path. Thus, adversaries do not know where they are and cannot hijack the legitimate applications.

B. FILE CONFIGURATION

By setting the 4 flags differently, we can configure files for different purposes. Table 2 summarizes them.

Regular files/directories do not have flags of PhantomFS. Any user can access them through the regular interface without reporting their access. They can also be accessed through the hidden interface.

Fake files/directories have flags rwf. They are only meant to allure adversaries and not actually in use. Existing honeyfiles [2], [3] fall under this category. Unlike existing honeyfiles, however, the fake files/directories in PhantomFS are hidden from the hidden interface. Thus, the fake files/directories do not generate false alarms because legitimate applications do not access them.

Some system files may be allowed to be read by any users but not allowed to be overwritten. For example, shared objects (dynamic link libraries) can be used by any applications, but should be updated only by a legitimate updater. If flag w is set to these files, it is reported to the administrator if adversaries try to modify them. We can use existing monitoring utilities (e.g. inotify of Linux) to monitor file access. However, existing utilities generate false alarms if the legitimate updater updates them. In contrast, PhantomFS does not generate false alarms because the legitimate updater updates them through the hidden interface.

We may use real files as decoys to allure adversaries. One potential problem of fake files is its stale metadata. If the metadata (e.g. last access time) has not been updated for a long time, adversaries may notice it even before opening the fake file. Thus, if a decoy file is alive through the hidden interface, it is more likely to succeed in alluring adversaries. However, the decoy file should not contain any sensitive information because adversaries can still read the file.

C. SCENARIOS

In this subsection, we give examples of how to use PhantomFS to thwart malicious users who have gained access to a host. Even though they login the host without being detected by intrusion detection systems, their malicious activities can be thwarted by PhantomFS.

1) RECONNAISSANCE

It is reported that the most common activities of malicious users when they login include checking and deleting the command history and exploring the system by reading system files [9]. It is often called a reconnaissance phase where the adversaries collect information about the system to figure out how to compromise it. Therefore, if we set flag r to those files that adversaries are likely to read, e.g. command history, they will be detected at the reconnaissance phase. We may create fake files (with flag f) with an appealing name, or use real system files as decoys. If we use fake files, we need to keep updating their metadata so that they may look alive.

2) RANSOMWARE

Ransomware is a type of malware that holds system or user files as hostage and asks for a ransom to regain access [13]. Ransomware typically searches the disk to find victim files and encrypt them. We can utilize PhantomFS to detect ransomware at the searching step. We may detect ransomware when files are being encrypted, but it would be better to detect it even before it starts encryption. By setting flag r to directories that ransomware is likely to search, ransomware can be detected. For example, many types of ransomware especially target at user files. For these kinds of ransomware, we can set flag r to “Documents” and its sub-directories.
If ransomware reads these directories through the regular interface, it can be detected before it starts encryption.

3) ILLEGAL MODIFICATION TO SYSTEM FILES
As an intermediate step or a result of an attack, system files may be modified. Representative examples are DLL injection and log scrubbing attacks. Adversaries acquire the administrative privilege by exploiting vulnerabilities of the system and modify the system files. If we set flag w to those files, we can detect unauthorized modification as long as adversaries do not use the hidden interface. PhantomFS offers one more barrier to control modification to the system files in addition to the existing access control mechanisms.

4) DATA LOSS PREVENTION
PhantomFS can help prevent sensitive data from being leaked. By setting flag h to sensitive files, they are hidden from the regular interface. Adversaries, who are not aware of PhantomFS, cannot find sensitive files through the regular interface. However, if adversaries already know the location of the sensitive files, adversaries can still access them because PhantomFS only hides them, but not controls access to them.

V. DESIGN AND IMPLEMENTATION
In this section, we explain how to implement the hidden interface, and how to accommodate them in more detail. Before presenting details of implementation, we summarize the design goals as follows.

- The hidden interface should not be too noticeable though we do not assume adversaries are aware of PhantomFS.
- The performance overhead in terms of throughput and delay should be minimized.
- Since we cannot assume that the source code of all applications is available, there must be a way to let the legitimate applications use the hidden interface without re-compiling them.

A. HIDDEN INTERFACE IMPLEMENTATION
Modern operating systems offer a set of system calls for file access. The hidden interface is an unknown system call to unauthorized users and applications. A straightforward way to implement the hidden interface is introducing a new system call. Though we assume adversaries are not aware of PhantomFS in this paper, we avoid using a new system call because it may be too noticeable. Instead, we implement it by interpreting an existing system call in a different way.

There are system calls that take a pointer to the user space as a parameter. The pointer is originally used to exchange data between the user-level application and the kernel. The read system call is a representative example. We use the read system call in our implementation, but we can use any system call that takes a user-space pointer.

By using the pointer, a data structure is sent to the kernel. We define a data structure for the hidden interface. It includes (1) signature, (2) request type, (3) request-specific parameters. Hidden interface implementation by using the read system call is illustrated in Figure 3.

The signature field is used to determine how the system call should be interpreted. The read system call can be used as a hidden interface, but also as a regular system call. Only if the signature matches, it is interpreted as a hidden interface. In our current implementation, we use 4 words for the signature. The signature field is cleared after processing the hidden interface to prevent the hidden interface from being called unintentionally. The user-space pointer can be freed and re-allocated to another application. The application may use the pointer when it calls the read system call in a regular way without initializing the memory region that the pointer refers to. In this case, the application may trigger the hidden interface unintentionally. To avoid it, the 4 words are erased after processing the hidden interface.

The next field is the request type. PhantomFS supports five types of requests: (1) read flags, (2) change flags, (3) read a directory, (4) read, and (5) write. The first two types of requests are used to read or change flags of a file. The remaining three types are used to implement the four flags. How to use the three types of requests (3, 4, and 5) will be explained in the following subsection.

The last field is the request-specific parameters. The request type of reading flags requires the file descriptor of the file as a parameter. That of changing flags requires the file descriptor and the new flags. The three remaining types of requests take the same parameters of their corresponding regular system calls.

B. MODIFICATION TO EXISTING SYSTEM CALLS
Modern operating systems provide system calls for file operations such as open, read, write, close, open a directory, read a directory, close a directory, etc. PhantomFS is prototyped in Linux. In Linux, the flags are checked by read, write, and getdents (for reading a directory) system calls. To implement PhantomFS, we introduce wrapper functions for them. Their pseudocode is shown in Algorithm 1.

The read system call is used for the hidden interface. Thus, when it is called, the signature is checked first. If it
In Linux, the inode interface is triggered unintentionally. If any file has flag $h$ set, a report is sent to the administrator. Then the original read system call is called as a regular interface. One of parameters of the read system call is the file descriptor to be read. If its flag $r$ is set, a report is sent to the administrator. Then the original read system call is called.

In the write system call, the original write system call is called after checking flag $w$. In the getdents system call, its original system call is called to read the list of files. Then flags of the files in the list are checked. If any file has flag $h$, the file is removed from the list so that the file should be hidden from the regular interface.

Function hidden_interface is used to process the requests of the hidden interface. If the request type is reading or changing flags, it is processed accordingly. If the request type is read or write, the original read or write system call is called. The parameters required to call the original system call are passed through the request data structure. If the request type is getdents, the original getdents system call is called, and then files with flag $f$ are removed from the list. Finally, the signature is erased to prevent the hidden interface from triggering unintentionally.

To store flags for each file, an additional field is required. In Linux, the inode structure must be the best candidate to add this field. In our prototype, however, to keep the file system compatible with the existing one, we did not change the inode structure. Instead, we implemented a separate lookup table in the kernel.

C. USING HIDDEN INTERFACE

Legitimate applications should access files through the hidden interface. However, it does not mean that all applications should do. Depending on the flag setting, there may be applications that are least likely to access files that trigger reporting. In other words, those applications that are likely to access files with flag $r$ or $w$, need to use the hidden interface to avoid false alarm.

Applications usually do not call system calls directly. They call wrapper functions provided by standard libraries. If the standard libraries are linked dynamically, they can be replaced easily by modifying the path to the linked library. If they are linked statically, the source code of the application is required to use the hidden interface. For PhantomFS, we provide modified libraries that use the hidden interface for those applications linked with the libraries dynamically. The modified libraries should locate in a hidden directory.

If the file name of the modified library remains same, only its path needs to be changed. In case of Linux, it can be done by changing the LD_LIBRARY_PATH environment variable. If the file name is changed, the executable of applications needs to be modified by a binary editor. In our prototype, we use vim to edit an executable. Figure 4 shows a screen capture of an executable when it is opened with vim. The library file names are shown as plain text such as libselinux.so.1 and libc.so.6. We can easily change them by using the standard commands of vim.

Figure 5 shows the detailed flowchart for an example of calling fwrite from an application. It shows operations of the library and the kernel. Note that it does not show all control paths, but only relevant ones.

Figure 5(a) shows the flowchart when the original unmodified kernel is employed. When the application calls fwrite, which is implemented in the standard library, the write
system call is called. The library also does other tasks, but they are omitted for brevity.

When PhantomFS is employed, the kernel is modified as explained in Section V-B. Those applications that use the regular interface are linked to the original library. Thus, when fwrite is called, the write system call is called. In the PhantomFS kernel, flag w is checked before executing the original write system call. If it is set, a report is sent to the administrator.

If an application is supposed to use the hidden interface, it is linked to the modified library. When fwrite is called, it is implemented using the hidden interface. To use the hidden interface, the data structure described in Section V-A is prepared and the read system call is called, instead of the write system call because the hidden interface is implemented through the read system call. The read system call works as explained in Section V-B.

VI. EXPERIMENTS

We prototype PhantomFS by modifying Linux 5.0.7. We run experiments on VirtualBox 5.2.26 running on a desktop with Intel i7, 32 GB main memory and 512 GB SSD. We ran Ubuntu 18.04 on the virtual machine with 4 GB memory and 50 GB virtual disk. The machine specification is summarized in Table 3.

We use a disk benchmark utility, dd, which is a part of coreutils 8.31, to measure the overhead of read and write operations. To measure the overhead of reading a directory, we measure the execution time of the utility, ls, which is also a part of coreutils 8.31. We modify glibc 2.29 which is linked with legitimate applications so that they can use the hidden interface. For changing flags of PhantomFS, we have developed a utility that sends a request of changing flags to the kernel through the hidden interface.

In this section, we demonstrate that PhantomFS incurs minimal overhead to disk I/O, successfully detects attacks, and generates no false alarm.

A. PERFORMANCE OVERHEAD

The performance overhead of the read system call is caused by checking the hidden interface and the flag. As shown in Algorithm 1, when the read system call is called, the buffer is read to check whether it is called for the hidden interface or not. Copying the buffer and comparing the signature incur performance overhead. If it is not called for the hidden interface, the flag is checked, which also incurs overhead.
FIGURE 6. Comparison of read and write throughput. There is no significant difference in throughput through the original system call (‘Original’), the modified regular interface of PhantomFS (‘Regular’), and the hidden interface of PhantomFS (‘Hidden’).

TABLE 3. Machine specification used for experiments.

| Host machine       | Processor | Memory | Disk        | Operating system |
|--------------------|-----------|--------|-------------|------------------|
|                    | Intel i7  | 32 GB  | 512 GB SSD  | Windows 10       |

| Virtual machine    | Virtualizer | Memory | Disk | Operating system | Kernel  |
|--------------------|-------------|--------|------|------------------|---------|
|                    | VirtualBox 5.2.26 | 4 GB   | 50 GB| Ubuntu 18.04     | Linux 5.0.7 |

To read a file through the hidden interface, the read system is called with the signature. If the signature matches, the hidden interface is processed. The read request is handled by calling the original read system call. Thus, handling the hidden interface incurs additional overhead to reading a file through the hidden interface.

The performance overhead of the write system call is caused by checking the flag. To write a file through the hidden interface, the read system call is used. It is called with the signature and the data structure for the write request. The hidden interface processes the request by calling the original write system call. The performance overhead of writing a file through the hidden interface is caused by handling the hidden interface.

Figure 6 shows the throughput of reading and writing files measured by dd and Figure 7 shows the delay of them. We vary the size of a request from 4 KB to 16 MB while the total amount of data is kept the same (4 GB). The number of requests is $T/S$, where $T$ is the total data (4 GB), and $S$ is the size of a request (from 4 KB to 16 MB). We record the throughput and delay reported by dd 100 times. Figure 6 and 7 show the average of 100 experiments by a thick histogram bar, and the maximum and minimum by a narrow error bar. We compare the throughput and delay of the original file system without PhantomFS (‘Original’), those of PhantomFS using the regular interface (‘Regular’), and those of PhantomFS using the hidden interface (‘Hidden’). In case of delay, we normalize the maximum, minimum, and average of Regular and Hidden to the average of Original because the delay varies heavily with the request size.

It is expected that PhantomFS incurs overhead in reading and writing data files, but what we observe is that there is no significant difference in throughput and delay as shown in Figure 6 and 7. It means that the overhead of additional code for PhantomFS has minimal impact on the overall performance of accessing data files.

The performance overhead of the getdents system call is caused by removing the hidden files. As shown in Algorithm 1, when the getdents system call is called, the original system call is called first to get the list of all files. Then the hidden files are removed by checking their flag. Precisely, files without $h$ and $f$ are copied to the buffer. It incurs additional memory copy. The number of additional memory copy is proportional to the number of files without $h$ and $f$. Thus, the overhead of reading a directory increases with the number of unhidden files.

Figure 8 compares the throughput of ls utility and Figure 9 does the delay. The throughput is measured in the number files per second. The delay is normalized to the average of Original. For Figure 8(a), we generate 100 test files, and set $h$ and $f$ flags of 0%, 25%, 50%, 75%, and 100% out of 100 test files. The throughput of Regular and Hidden slightly increases with the percentage of hidden files because the number of additional memory copy decreases. The throughput of Original does not change. When the total number of files increases to 1000, this trend becomes more evident, as shown in Figure 8(b). When there is no hidden files (0%), the throughput of Regular and Hidden is 10.82% and 9.53% lower than that of Original, respectively. As the percentage of hidden files increases, their throughput increases while that of Original remains same. We can observe the same trend in delay as shown in Figure 9.

B. IMPLEMENTATION OVERHEAD

To estimate the overhead of implementation, we measure the number of lines of code modified to implement PhantomFS. In total, we modify 356 lines of Linux kernel 5.0.7 and 76 lines of glibc 2.29. Table 4 breaks them down to files. Overall, the number of modified lines is minimal, which means that the implementation overhead of PhantomFS is affordable.
C. THwarting Malicious Activities

We test the following scenarios of malicious activities to compare PhantomFS with the honeyfile technique [3], which is the closest existing work to PhantomFS. Table 5 summarizes the results.

1) RECONNAISSANCE

**Test scenario:** At the reconnaissance phase, adversaries try to collect information about the host, once they successfully invaded the host. Note that PhantomFS targets at adversaries who have already gained access to a host. As mentioned in Section IV-C, adversaries collect information by reading system files. Thus, if we set flag $r$, to those files that are likely to be accessed by adversaries, we can detect them. We use the command history file for this test scenario because it is one of the most popular target files that adversaries read when they login [9]. The test scenario of PhantomFS is as follows.

- The administrator creates the **victim** user account for testing.
- The administrator sets flag $r$ to `.bash_history` in the home directory of **victim**.
- The adversary gains access to the host.
- The adversary reads `.bash_history` by using utility `cat`, which locates at `/bin` and uses the regular interface.
Instead of PhantomFS, if the existing honeyfile is employed, we cannot use the existing file (in this scenario, .bash_history) as a decoy. To allure the adversary, we create a decoy file whose name is similar. The test scenario of the honeyfile is as follows.

- The administrator creates the victim user account for testing.
- The administrator places a decoy file whose name is similar in the home directory of the victim.
- The adversary gains access to the host.
- The adversary logs in.
- The adversary reads the decoy file using utility cat.

**Result:** Both PhantomFS and the honeyfile detect the adversary successfully under the scenario described above. However, the honeyfile technique generates false alarms whenever the legitimate indexing service reads .command_history by mistake. In contrast, PhantomFS avoids generating false alarms by letting the legitimate user read the directory through the hidden interface.

### 2) RANSOMWARE

**Test scenario:** We test a ransomware sample, CryptSky [26], which starts searching victim files from /home. If we set flag r to /home directory, and we can observe an alarm is raised when CryptSky starts searching for files. Unlike existing signature or behavior based techniques [13], [27]–[30], ransomware can be detected even before it starts encryption.

To test false alarm, we use the modified ls utility which uses the hidden interface to read directories. Since it reads directories through the hidden interface, we observed no false alarm when we run it to read the /home directory.

The test scenario of PhantomFS is as follows.

- The administrator creates the victim user account for testing.
- The administrator sets flag r to the /home directory.
- The adversary gains access to the host.
- The adversary runs CryptSky.
- The legitimate user logs in.
- The legitimate user reads a script in a hidden directory (in this scenario, it is in /123qweASD), which changes the default search path to /123qweASD/bin.
- The legitimate user reads the /home directory by using ls, which locates at /123qweASD/bin and uses the hidden interface.

For the honeyfile technique, we place a decoy file secret.doc at /home. The test scenario of the honeyfile is as follows.

- The administrator creates the victim user account for testing.
- The administrator places a decoy file, secret.doc at the /home directory.
- The adversary gains access to the host.
- The adversary runs CryptSky.
- The legitimate user logs in.
- The legitimate user reads the decoy file using the word processor.

**Result:** Both PhantomFS and the honeyfile detect the ransomware successfully under the scenario described above. However, the honeyfile technique generates false alarms when the legitimate user reads the decoy file while PhantomFS does not. PhantomFS avoids generating false alarms by letting the legitimate user read the directory through the hidden interface.

### 3) LOG SCRUBBING

**Test scenario:** Adversaries may want to hide what they did by erasing their traces in log files. To detect such log scrubbing attacks, we test setting w to syslog. Reading the log file does not generate any alarm, but if a user (potentially adversary) tries to edit the file, we observe an alarm is generated.

The legitimate service, rsyslogd needs to write data to the log file. We edit its executable to change its linking library to what we provide. The library we provide uses the hidden interface. Thus, the legitimate service does not generate false alarms.

The legitimate user reads the log file by using utility cat. We edit its executable to change its linking library to what we provide. The library we provide uses the hidden interface. Thus, the legitimate service does not generate false alarms.

**Result:** Both PhantomFS and the honeyfile technique detect the ransomware successfully under the scenario described above. However, the honeyfile technique generates false alarms when the legitimate user reads the decoy file while PhantomFS does not. PhantomFS avoids generating false alarms by letting the legitimate user read the directory through the hidden interface.

### 3) LOG SCRUBBING

**Test scenario:** Adversaries may want to hide what they did by erasing their traces in log files. To detect such log scrubbing attacks, we test setting w to syslog. Reading the log file does not generate any alarm, but if a user (potentially adversary) tries to edit the file, we observe an alarm is generated.

The legitimate service, rsyslogd needs to write data to the log file. We edit its executable to change its linking library to what we provide. The library we provide uses the hidden interface. Thus, the legitimate service does not generate false alarms.

**Result:** Both PhantomFS and the honeyfile technique detect the ransomware successfully under the scenario described above. However, the honeyfile technique generates false alarms when the legitimate user reads the decoy file while PhantomFS does not. PhantomFS avoids generating false alarms by letting the legitimate user read the directory through the hidden interface.
J. Lee et al.: PhantomFS: File-Based Deception Technology for Thwarting Malicious Users

Result: PhantomFS successfully detects the malicious activity of the log scrubbing attack and does not generate false alarms because the legitimate users do not use the hidden interface to update the log file. Furthermore, if the legitimate user reads the log file, no false alarm will be generated.

4) DATA LOSS PREVENTION

Test scenario: We can hide data files by setting flag h. The hidden data files cannot be seen through the regular interface. We test it by using the unmodified `ls` utility, which uses the regular interface. In contrast, the legitimate user can find it by using the modified `ls` utility.

The test scenario is as follows. Note that this scenario cannot be supported by the honeyfile technique.

- The administrator creates the victim user account for testing.
- The administrator sets flag h to the top_secret.doc file in the /home/victim/Documents directory.
- The adversary gains access to the host.
- The adversary reads the /home/victim/Documents directory by using `ls`, which locates at /bin and uses the regular interface.
- The legitimate user logs in.
- The legitimate user runs a script in /123qweASD, which changes the default search path to /123qweASD/bin.
- The legitimate user reads the /home/victim/Documents directory by using `ls`, which locates at /123qweASD/bin and uses the hidden interface.

Result: PhantomFS successfully prevents the adversary from finding top_secret.doc because the adversary reads the directory through the regular interface. The legitimate user can find it by using the modified `ls` which uses the hidden interface.

VII. CONCLUSION

In this paper, we propose PhantomFS which is a file-based deception technology to thwart malicious users who have gained access to a host. It introduces the concept of using a hidden interface to address the issue of false alarm. Through the hidden interface, the legitimate users and applications can avoid accessing fake files. It can also be utilized to hide sensitive files from the regular interface to prevent malicious users from accessing sensitive files. We prototype PhantomFS in Linux and demonstrate that the overhead of reading and writing files, and reading a directory is negligible. By experiments, we also validate that PhantomFS achieves 100% detection rate and 0% false alarm rate for the given test scenarios. In our future work, we will address how to prevent intelligent adversaries who are aware of PhantomFS and try to nullify it.

Generated by IEEEtran.bst, version: 1.13 (2008/09/30)

REFERENCES

[1] Wikipedia. Honeypot (Computing). Accessed: 2020. [Online]. Available: https://en.wikipedia.org/wiki/Honeypot_(computing)
[2] D. Fraunholz, S. D. Anton, C. Lippis, D. Reti, D. Krohmer, F. Pohl, M. Tammen, and H. D. Schotten, “Demystifying Deception Technology: A Survey,” 2018, arXiv:1804.06196. [Online]. Available: http://arxiv.org/abs/1804.06196
[3] J. Yuill, M. Zappe, D. Demning, and F. Feer, “Honeyfiles: Deceptive files for intrusion detection,” in Proc. 5th Annu. IEEE SMC Inf. Assurance Workshop, Jun. 2005, pp. 116–122.
[4] L. Spitzner, “The honeynet project: Trapping the hackers,” IEEE Secur. Privacy, vol. 1, no. 2, pp. 15–23, Mar. 2003.
[5] D. Fraunholz, D. Krohmer, F. Pohl, and H. D. Schotten, “On the detection and handling of security incidents and perimeter breaches—A modular and flexible honeypot based framework,” in Proc. 9th IFIP Int. Conf. New Technol., Mobility Secure (NTMS), Feb. 2018, pp. 1–4.
[6] M. Bercovitch, M. Renford, L. Hassan, A. Shabtal, L. Rokach, and Y. Elovici, “HoneyGen: An automated honeypots generator,” in Proc. IEEE Int. Conf. Intell. Secur. Informat., Jul. 2011, pp. 131–136.
[7] A. Juels and R. L. Rivest, “Honeypots: Making password-cracking detectable,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), New York, NY, USA: ACM, 2013, pp. 145–160, doi: 10.1145/2508859.2516671.
[8] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, “From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), New York, NY, USA: Association for Computing Machinery, 2014, pp. 942–953, doi: 10.1145/2660267.2660329.
[9] B. M. Bowen, S. Hershop, A. D. Keromytis, and S. J. Stolfo, “Baiting inside attackers using decoy documents,” in Security and Privacy in Communication Networks, Y. Chen, T. D. Dimitriou, and J. Zhou, Eds. Berlin, Germany: Springer, 2009, pp. 51–70.
[10] B. Whitham, “Canary files: Generating fake files to detect critical data loss from complex computer networks,” in Proc. 2nd Int. Conf. Cyber Secur., Cyber Peacefare Digit. Forensic (CyberSec), Mar. 2013, pp. 1–10.
[11] M. Lazarov, J. Ounislapo, and G. Stringhini, “Honey sheets: What happens to leaked google spreadsheets?” in Proc. 9th Workshop Cyber Secur. Experimentation Test (CSET), Austin, TX, USA: USENIX Association, 2016, pp. 1–8. [Online]. Available: https://www.usenix.org/conference/cset16/workshop-program/presentation/lazarov
[12] B. Whitham, “Automating the generation of enticing text content for high-interaction honeyfiles,” in Proc. Hawaii Int. Conf. Syst. Sci., Jan. 2017, pp. 1–10.
[13] C. Moore, “Detecting ransomware with honeypot techniques,” in Proc. Cybersecurity Cyberforensics Conf. (CCC), Aug. 2016, pp. 77–81.
[14] S. Laurén, S. Rauti, and V. Leppänen, “An interface diversified honeypot for malware analysis,” in Proc. 10th Eur. Conf. Softw. Archit. Workshops (ECSAW), New York, NY, USA: ACM, 2016, pp. 29–39, doi: 10.1145/2993412.2993417.
[15] S. Laurén, P. Maki, S. Rauti, S. Hosseinzead, S. Hyyrynsalmi, and V. Leppänen, “Symbol diversification of linux binaries,” in Proc. World Congr. Internet Secur. (WorldCIS), Dec. 2014, pp. 74–79.
[16] N. C. Rowe, E. J. Custy, and B. T. Duong, “Defending cyberspace with fake honeypots,” J. Comput., vol. 2, no. 2, pp. 25–36, Nov. 2009.
[17] J. Shi, Y. Yang, C. Li, and X. Wang, “SPEMS: A stealthy and practical execution monitoring system based on VMI,” in Cloud Computing and Security, Z. Huang, X. Sun, J. Luo, and J. Wang, Eds. Cham, Switzerland: Springer, 2015, pp. 380–389.
[18] V. E. Urias, W. M. S. Stout, and C. Loverro, “Computer network deception as a moving target defense,” in Proc. Int. Carnahan Conf. Secur. Technol. (ICCST), Sep. 2015, pp. 1–6.
[19] K. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, “Scalability, fidelity and stealth in the dkraufv dynamic malware analysis system,” in Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC), New York, NY, USA: ACM, 2014, pp. 386–395, doi: 10.1145/2664243.2664252.
[20] M. Beham, M. Vlad, and H. P. Reiser, “Intrusion detection and honeypots in nested virtualization environments,” in Proc. 43rd Annu. IEEE/FIP Int. Conf. Dependable Syst. Netw. (DSN), Jun. 2013, pp. 1–6.
[21] S. Ganapathy, P. Yogesh, and A. Kannan, “An intelligent intrusion detection system for mobile ad-hoc networks using classification techniques,” in Advances in Power Electronics and Instrumentation Engineering, V. V. Das, N. Thakkachen, and N. C. Deb Nath, Eds. Berlin, Germany: Springer, 2011, pp. 117–122.
[22] D. S. Vijayakumar and S. Ganapathy, “Machine learning approach to combat false alarms in wireless intrusion detection system,” Comput. Inf. Sci., vol. 11, no. 3, pp. 67–81, Jul. 2018.
J. Lee et al.: PhantomFS: File-Based Deception Technology for Thwarting Malicious Users

[23] G. Creech and J. Hu, “A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns,” IEEE Trans. Comput., vol. 63, no. 4, pp. 807–819, Apr. 2014.

[24] S. N. Chari and P.-C. Cheng, “BlueBox: A policy-driven, host-based intrusion detection system,” ACM Trans. Inf. Syst. Secur., vol. 6, no. 2, pp. 173–200, May 2003. doi: 10.1145/762476.762477.

[25] D.-Y. Yeung and Y. Ding, “Host-based intrusion detection using dynamic and static behavioral models,” Pattern Recognit., vol. 36, no. 1, pp. 229–243, Jan. 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320302000262

[26] Cryptsky. Accessed: 2018. [Online]. Available: https://github.com/deadPix3l/CryptSky

[27] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, “Cryptolock (and drop it): Stopping ransomware attacks on user data,” in Proc. ICDCS, Jun. 2016, pp. 1–10.

[28] E. Kolodenker, W. Koch, G. Stringhini, and M. Egele, “PayBreak: Defense against cryptographic ransomware,” in Proc. ACM Asia Conf. Comput. Commun. Secur. (ASIA CCS), 2017, pp. 599–611.

[29] J. Huang, J. Xu, X. Xing, P. Liu, and M. K. Qureshi, “FlashGuard: Leveraging intrinsic flash properties to defend against encryption ransomware,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS), 2017, pp. 2231–2244.

JUNGHEE LEE (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, in 2000 and 2003, respectively, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, in 2013. From 2003 to 2008, he worked on electronic system level design of mobile system-on-chips with Samsung Electronics. From 2014 to 2019, he was an Assistant Professor with the Department of Electrical and Computer Engineering, The University of Texas at San Antonio. He has been with the School of Cybersecurity, Korea University, since 2019. His research interests include secure design or hardware-assisted security of processors, non-volatile memory and storage, and dedicated hardware.

JIONE CHOI received the B.S. degree in computer science from Dongduk University, in 2019. She is currently pursuing the M.S. degree in cybersecurity with Korea University, South Korea. From May 2018 to July 2019, she worked as an Intern with an AI company called Furiosa AI, where she was a Profiler Programmer of the Neural Processing Unit. Her research interests include Linux kernel, and file system and security such as intrusion detection and honey pot.

GYUHO LEE received the B.S. and M.S. degrees in computer engineering and information communication from Inha University, in 2004 and 2012, respectively. He has been with LIG NEX1 Company, Ltd., South Korea, since 2007. His research interests include software security of weapon systems, anti-tampering platform, secure development process, and cyber kill chain.

SHIN-WOO SHIM received the B.S. degree in computer science and engineering from POSTECH, in 2007, and the M.S. degree in information security from Korea University, in 2019. He has been working with LIG NEX1 Company, Ltd., South Korea, since 2007. His research interests include command and control in cyber warfare, mission impact analysis, and course of action in cyber warfare.

TAEKYU KIM received the B.S. degree in computer science and engineering from Chung-Ang University, in 2000, and the M.S. and Ph.D. degrees in electrical and computer engineering from The University of Arizona, in 2006 and 2008, respectively. He was working with the Software Techniques Team, SK E&C, in 2008. He has been with LIG NEX1 Company, Ltd., South Korea, since 2010. His research interests include cyber operation systems, training systems, and cyber infra systems.

* * *