IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 30, 2020, accepted October 22, 2020, date of publication October 28, 2020,
date of current version November 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3034443

PhantomFS-v2: Dare You to Avoid This Trap

JIONE CHOI[1], HWIWON LEE[1], YOUNGGI PARK[1], HUY KANG KIM[1], (Member, IEEE), JUNGHEE LEE[1], (Member, IEEE), YOUNGJAE KIM[2], (Member, IEEE), GYUHO LEE[3], SHIN-WOO SHIM[3], AND TAEKYU KIM[3]

[1]School of Cybersecurity, Korea University, Seoul 02841, South Korea
[2]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea
[3]Cyber Warfare Research and Development Laboratory, LIG Nex1, Seongnam 13488, South Korea

Corresponding author: Junghee Lee (j_lee@korea.ac.kr)

ABSTRACT It has been demonstrated that deception technologies are effective in detecting advanced persistent threats and zero-day attacks which cannot be detected by traditional signature-based intrusion detection techniques. Especially, a file-based deception technology is promising because it is very difficult (if not impossible) to commit an attack without reading and modifying any file. It can play as an additional security barrier because malicious file access can be detected even if an adversary succeeds in gaining access to a host. However, PhantomFS still has a problem that is common to deception technologies. Once a deception technology is known to adversaries, it is unlikely to succeed in alluring adversaries. In this paper, we classify adversaries who are aware of PhantomFS according to their knowledge level and permission of PhantomFS. Then we analyze the attack surface and develop a defense strategy to limit the attack vectors. We extend PhantomFS to realize the strategy. Specifically, we introduce multiple hidden interfaces and detection of file execution. We evaluate the security and performance overhead of the proposed technique. We demonstrate that the extended PhantomFS is secure against intelligent adversaries by penetration testing. The extended PhantomFS offers higher detection accuracy with lower false alarm rate compared to existing techniques. It is also demonstrated that the overhead is negligible in terms of response time and CPU time.

INDEX TERMS Deception technology, file system, honeypot.

I. INTRODUCTION

A honeypot is a fake host that allures adversaries so that their activities can be observed and analyzed [1]. It has been extended as the concept of deception technology and applied to various entities of systems [2]. Fake database [3], [4], password [5], account [3], and patch [6] are used to allure adversaries [7]. They can play as an additional security barrier to thwart adversaries who succeed in invading a host evading intrusion detection or prevention systems. They are known to be more effective in insider threats, social engineering, and 0-day attacks than traditional perimeter or signature-based intrusion detection and anomaly detection techniques [7].

The file-based deception technology is one of such deception technologies. If malicious users or applications access fake files, it is reported as a potential intrusion to administrators. It is very hard (if not impossible) to commit attacks without accessing a single file. Adversaries often read system files to gather information about the host and make changes to system files as a result or an intermediate step of an attack. Therefore, monitoring file access is expected to be very effective in detecting misbehavior of malicious users. It can be located close to those files under protection, and detects/prevents malicious file access even if traditional signature-based intrusion detection systems fail to detect them.

The file-based deception technology, however, may suffer from false alarms [8]. Legitimate users may access fake files by mistake and so do legitimate applications (e.g. file indexing). PhantomFS [9] addresses this issue by introducing a hidden interface. The hidden interface is known only to legitimate users and applications. Alarms are not generated if files are accessed via the hidden interface. Adversaries, who do not know the hidden interface and use the regular interface, generate alarms if they access fake files. In contrast, legitimate users do not generate false alarms because they use the hidden interface.

PhantomFS, however, still has a limitation. It is effective in alluring adversaries who are not aware of it. However, if adversaries know about it, they will try to avoid or nullify

The associate editor coordinating the review of this manuscript and approving it for publication was Zhitao Guan.

it. In fact, this is a common issue across all kinds of deception technologies. Once adversaries become aware of them, adversaries can find a way to avoid them [7]. In order for deception technologies to be used in practice, there must be countermeasures to those who are aware of them.

To address this concern, we proposed PhantomFS-v2, an extension of PhantomFS. It offers countermeasures to intelligent adversaries who are aware of the existence of PhantomFS-v2. To achieve this goal, we analyze the attack surface of the original PhantomFS in this paper. We discuss what adversaries would do if they know the existence of PhantomFS. From this analysis, we identify potential attack vectors. We add countermeasures to prevent them, such as multiple hidden interfaces, flagging executables, modification to `proc` file system, etc. By the penetration testing, we demonstrate the effectiveness of PhantomFS-v2. PhantomFS-v2 covers wider range of attack scenarios with no false alarms than the original PhantomFS and existing similar techniques.

This paper is organized as follows. We briefly introduce the original PhantomFS and discuss the related works in Section II. After defining the threat model and analyzing the attack surface in Section III, we present our defense strategy in Section IV and illustrate it with an example in Section V. In Section VI, we explain how to realize the defense strategy by extending PhantomFS. The experimental results are presented in Section VII, followed by conclusions in Section VIII.

II. BACKGROUND AND RELATED WORKS

A. PhantomFS

The goal of PhantomFS [9] is to allure adversaries who have already gained access to a host without being detected by traditional intrusion detection systems. PhantomFS offers an additional security barrier to thwart such adversaries.

PhantomFS offers a hidden interface besides the regular interface. As illustrated in Figure 1, the legitimate users and applications use the hidden interface, while adversaries, who are not aware of PhantomFS, access files through the regular interface. If adversaries access a flagged file through the regular interface, it is reported as a potential intrusion to the administrator. Administrators can hide sensitive files from the regular interface , which can be accessed only via the hidden interface. Through the hidden interface, legitimate users can access files without triggering an alarm, which reduces the chance of false alarms.

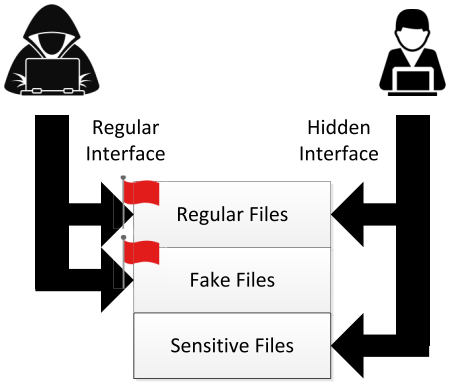


FIGURE 1. PhantomFS deceives adversaries by providing a hidden interface. Accessing flagged files via the regular interface is reported as a symptom of intrusion. Reprinted from [9], Copyright 2020 by IEEE.

The original PhantomFS maintains four additional flags for each file, as shown in Table 1. Flag *h* is to hide a file from the regular interface. Sensitive files can be hidden by using this flag. If flag *r* or *w* is set, it is reported to the administrator if the file is read or modified through the regular interface. Flag *f* is to hide a file from the *hidden interface*. The flags can also be used for directories.

TABLE 1. Four additional flags used for the original PhantomFS.

Flag	Meaning
h	Hidden from the regular interface
r	Report if this file is read
w	Report if this file is written
f	Hidden from the hidden interface

PhantomFS works independently from the existing access control mechanisms. Even though a file with flag *h* is not shown, it is still accessible if its path is known. Flags *r* and *w* are orthogonal to the access permission. Setting those flags does not prevent adversaries from accessing the file.

The four flags are checked by their corresponding system calls. In Linux, the list of files is read through `getdents` system call, and a file is read and modified by `read` and `write` system calls. Before `getdents` returns the list of files, it removes those files with flag *h* to hide them. The `read` and `write` system calls send an alarm if the requested file has flag *r* or *w*.

The hidden interface is implemented by using the `read` system call. The caller provides a pointer to a buffer, which is originally used to receive data from the kernel. When the `read` system call is used as a hidden interface, a data structure is stored in the buffer. When the `read` system call is called, it reads the first four words of the buffer. If they match with the pre-defined signature, it means the system call is called as a hidden interface. Otherwise, it is called as a regular `read` system call.

The hidden interface supports `getdents`, `read`, and `write` requests, which correspond to the system calls. In addition, it also processes the request of reading or changing PhantomFS flags. The information required to process the request is stored in the buffer after the signature. In summary, the pseudocode of the modified system calls is given in Algorithm 1.

The legitimate applications need to be modified to use the hidden interface. Applications usually call wrapper functions in the library instead of directly calling system calls. Most of Linux applications use the standard `glibc` library to access files. Thus, a modified library is provided, which uses the hidden interface, and the legitimate applications need to be

Algorithm 1 Pseudocode of the Hidden Interface and Modified System Calls

```
1:  procedure GETDENTS
2:      call the original getdents system call
3:      remove files whose flag h is set
4:  procedure READ
5:      if the signature matches then
6:          call the hidden_interface function
7:          return
8:      if flag r is set to this file then
9:          send a report
10:     call the original read system call
11: procedure WRITE
12:     if flag w is set to this file then
13:         send a report
14:     call the original write system call
15: procedure HIDDEN_INTERFACE
16:     if the request type is reading or changing flags then
17:         process the request
18:     if the request type is getdents then
19:         call the original getdents system call
20:         remove files whose flag f is set
21:     if the request type is read then
22:         call the original read system call
23:     if the request type is write then
24:         call the original write system call
25:     erase the signature
```

linked with it. The standard library is usually linked dynamically. In this case, the library can be changed by changing the path of the library.

The legitimate users should use the legitimate applications which use the hidden interface. To prevent the legitimate users from triggering false alarms by mistake, the following approach is used. The legitimate applications locate in a hidden directory keeping the same name. For example, the legitimate `ls` may locate in a hidden directory, `/PhantomFS_HiddenDir_123qweASD`. The original `ls`, which uses the regular interface, still locates in `/bin`. If the user runs `ls`, the original one is used, which may trigger false alarms. However, if the user changes the search path to the hidden directory, the user can use `ls` as if nothing has changed. The hidden directory should have flag *h* so that it can be hidden from the regular interface. Thus, those users who know the correct path of the hidden directory can access it, whereas adversaries who do not know it cannot. The modified library should be also in the hidden directory. The hidden path plays as a similar role with a password. The legitimate user should keep it safely.

B. RELATED WORKS

The deception technology stems from a honeypot, which is a fake host used to allure adversaries [1]. It has been extended and applied to various entities of systems [3]–[7]. The file-based deception technology is one of them, which detects adversaries when they access fake files [8]–[12]. Ransomware is a good target for the file-based deception technology because ransomware is likely to access fake files while it is searching for victim files [13].

The common issue of deception technologies is that they are not effective any longer if adversaries become aware of the deception technologies [7]. One approach to address this issue can be the moving target defense. This technique is originally intended to make it harder for adversaries to find the correct target by constantly changing the configuration or the attack surface [14]. It has been extensively studied including randomization of network configuration [15]–[19], hosts [20], [21], operating system services [22], [23], compilers [24], instruction sets [25], [26], address space [27], and database queries [28].

In case of the file-based deception technology, HoneyGen is proposed, which automatically generates decoy files from the profiles of real files [29] so that decoy files may look as close as possible to real files. HoneyGen is not specifically intended to address this issue, but makes it harder for adversaries to distinguish fake files from real files even if adversaries become aware of the existence of fake files.

In summary, existing techniques can be used to raise the bar of successful attacks for those adversaries who know the deception technology. We also employ this concept partially in PhantomFS-v2, but our ultimate goal is eliminating possibility of successful attacks even if adversaries become aware of PhantomFS-v2.

We position the proposed technique as another security barrier in addition to intrusion detection systems. Even if adversaries gain access to a host evading traditional intrusion detection systems (IDSs), they can still be detected when they access any decoy file. PhantomFS-v2 is a complementary solution to IDSs.

IDSs can be classified mainly into network-based IDSs (NIDSs) [30], [31] and host-based IDSs (HIDSs) [32]–[34]. NIDSs detect malicious activities by monitoring network packets while HIDSs monitors activities on the host. Typically, their detection algorithms are based on matching the signature of known malicious activities [33] or identifying abnormal activities deviated from the known normal behavior [34]. Intelligent IDSs often employ intelligent agents, neural networks, genetic algorithms, fuzzy sets, particle swarm and soft computing techniques [35]. One of well known techniques for HIDSs is file integrity checking. It detects unauthorized modification to files under protection by measuring integrity metric, which is typically a hash value of a file. The integrity metric of a file is measured and compared to a reference integrity metric which had been measured a priori. If the integrity metric mismatches, the modification to the file is detected.

When virtual machines or containers are employed, different users have different file views. However, their original intention is not on deceiving adversaries nor intrusion

detection. It is usually not supported to configure individual files for different file views.

III. ATTACK SURFACE

In this section, we analyze the potential attack surface if adversaries are aware of PhantomFS-v2. To do so, we first classify adversaries according to their capability and then we discuss attack surfaces.

A. THREAT MODEL

PhantomFS-v2 aims at alluring adversaries who have evaded intrusion detection systems and successfully gained access to a host. They may acquire the administrative privilege by taking advantage of vulnerabilities. They may succeed in launching a shell and execute existing utilities. It is also possible for them to install a hacking tool on the host. They may inject malicious code to a legitimate application by exploiting its vulnerabilities.

We do not assume kernel-level malware (rootkits) because PhantomFS-v2 is implemented in the kernel. If a rootkit exists, the correct operation of PhantomFS-v2 cannot be guaranteed.

The original paper [9] assumes adversaries do not know anything about PhantomFS. They do not know the existence of the hidden interface and the location of the hidden path. Since legitimate applications and the modified library locate in the hidden path, the adversaries do not know where they are.

In this paper, we assume adversaries are aware of PhantomFS-v2. According to the knowledge and permission of PhantomFS-v2, we further classify attackers into four categories and name them as follows.

- **Outside Attackers** are aware of PhantomFS-v2 in general, but they do not know the specific configurations on the host they are accessing. The administrator configures flags of files and hides files to a hidden path. The outside attackers do not know how the administrator has configured them.
- **Impersonation Attackers** are aware of PhantomFS-v2 and impersonate a legitimate user. They have stolen credentials of a legitimate user including the password and the hidden path. Thus, they have the permission to access the hidden interface. However, they do not know the configurations.
- **Inside Attackers** know all the details of PhantomFS-v2 including the configurations, but do not have the permission to access the target host. They do not have a legitimate account and cannot access the hidden interface of the target host.
- **Traitors** know all the details and configurations of PhantomFS-v2 and have the permission. They are those attackers who access resources under their own control. They can be administrators or normal users. They may try to leak secrets, counterfeit or destroy information, and disable security solutions, which are under their control.

The focus of this paper is on the first three types of attackers because it is very difficult to distinguish the misbehavior of traitors from the normal behavior. In addition, we expect the motivation of traitors is much weaker than that of others. This is because the traitors cannot be free from the charge, if it turns out that cyber incident is involved with resources managed by the traitors.

B. ATTACK SURFACE

We have identified three potential targets of attacks: hidden path, hidden interface and evasion.

1) HIDDEN PATH

Adversaries may try to figure out the location of the hidden path. The hidden path is not seen via the regular interface, but still accessible if its correct path is known. Once it is known, adversaries can use the legitimate applications which locate there.

As explained in Section II, the hidden path plays a similar role with a password. Legitimate users should keep it safely. There are non-technical ways to steal it (e.g. social engineering), but they are out of scope from the discussion here.

If adversaries read the command history of legitimate users, they can find the hidden path. Legitimate users are required to run a script that changes the default search path to the hidden path. Since the script itself is hidden in the hidden path, the command history reveals the path.

Some system utilities (e.g. `ps` and `proc` file system) show the absolute path of running applications. By using one of them, adversary may figure out where the legitimate applications locate.

2) HIDDEN INTERFACE

Adversaries may steal the hidden interface that they are not supposed to use. It can be achieved by reverse engineering the code of the system calls. If they know which system call is used to implement the hidden interface, they may figure out where the code of that system call is in the main memory and how the system call works. They can also do this by analyzing the code of legitimate applications or the modified library.

Even if they cannot figure out how the hidden interface works, they can use it by hijacking legitimate applications. There may be legitimate system services and applications continuously running on the host. Some of them may need to use the hidden interface to avoid false alarm. Adversaries may inject malicious code to them by exploiting their vulnerabilities. The injected code can use the hidden interface if the victim application does.

3) EVASION

The other approach that the adversaries may try is to evade PhantomFS-v2. The implementation of the original PhantomFS checks flags only when the `read`, `write` and `getdents` system calls are called because they are standard interface to access files. If adversaries access files in a non-standard way, they may succeed without triggering

TABLE 2. Summary of the attack vectors and defense strategy.

Target	Attack vector	Defense	
Hidden path	Command history	Flagging the history file	Multiple hidden interface and flagging executables
	Utilities based on `proc`	Modification to `proc`	
Hidden interface	Reverse engineering	Diverse implementation of hidden interfaces	
	Control flow hijacking	Limiting the impact	
	Malware installation	Prohibiting, hiding, or flagging development tools	
Evasion	File access in a non-standard way	Checking flags in all system calls that can be used to access files	

alarm. For example, they may use `mmap` which maps files into memory so that the application can access files as if it accesses memory. In this way, adversaries can access files without calling `read` and `write` system calls.

Since PhantomFS-v2 is implemented in the file system, adversaries may try to evade PhantomFS-v2 by accessing the disk (raw block device) without going through the file system. If they access the raw block device directly, they can access files without triggering alarm. To do so, adversaries need to figure out the file system type and on-disk data structures in detail.

It is also possible for adversaries to disable reporting to administrators. In fact, it is a general attack surface that can be used for any security solutions. Though PhantomFS-v2 does not dictate a specific reporting mechanism, it may be disabled by adversaries.

IV. DEFENSE STRATEGY

In this section, we discuss how to defend against potential attacks analyzed in the previous section. Once adversaries succeed in gaining access to a host, they may run executables on the host to explore and damage the host. These activities can be detected by flagging executables. However, they may run legitimate executables if they succeed in stealing a hidden interface by impersonation or hijacking legitimate applications. By employing multiple hidden interfaces, the impact of stealing can be limited. Even if one hidden interface is exposed, those files flagged by other hidden interfaces are still protected. Though the resources available to adversaries are greatly limited by flagging executables and employing multiple hidden interfaces, adversaries can still try to nullify PhantomFS-v2 by using what is available. To further reduce the risk, we identify potential attack vectors and their defense strategies. Table 2 summarizes our defense strategy. We discuss its details in the following subsections.

A. COMMON DEFENSE STRATEGY

1) MULTIPLE HIDDEN INTERFACES

First of all, we need multiple hidden interfaces to defend against impersonation attackers who can use the hidden interface. If there is only one hidden interface, PhantomFS-v2 can be easily nullified by them. There must be more than one hidden interface, and they should be assigned to users according to their role or privilege.

As illustrated in Figure 2, the file view is different depending on the hidden interface. It is allowed to set flags to hidden

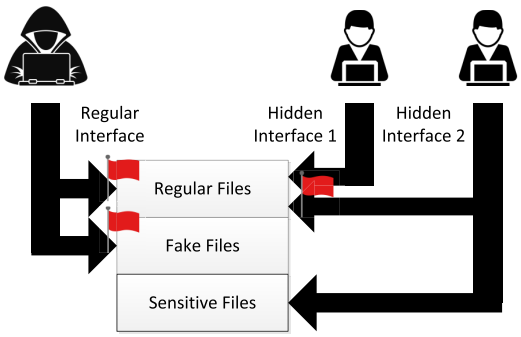


FIGURE 2. PhantomFS-v2 provides multiple hidden interfaces and allows to set flags to hidden interfaces.

interfaces. For example, when a flag is set to a file by a hidden interface, it means an alarm is triggered if the file is accessed through other hidden interfaces or the regular interface. Thus, even if adversaries succeed in impersonating a legitimate user, they are still limited on accessing files on the host.

In the approach using multiple interfaces, flag f is no longer necessary, and flags h, r, and w can be set for hidden interfaces as well. Table 5 shows an example of setting flags to different interfaces, which will be explained in detail later.

However, multiple hidden interfaces cannot defend against all types of attacks by impersonation attackers. Since they can use one of the hidden interfaces, they can still commit those attacks that are permitted by the hidden interface. To cope with this, the hidden interface should grant the least permission that is required for the user to perform the given task. This is the same philosophy with when using `sudo`. It should also be applied to applications when a hidden interface is assigned to an application because the application may be hijacked. In this way, we can limit the impact when one of the hidden interfaces is taken by adversaries.

2) FLAGGING EXECUTABLES

As discussed in Section II, adversaries may inject a malicious code to an application, install malware, or launch a shell. When they succeed in launching a shell, they can execute existing utilities to commit attacks. If we set a flag to system utilities that adversaries are likely to execute, we can detect their attacks. For this, we introduce a new flag, x. When flag x is set to an executable, an alarm is generated if it is executed.

Legitimate applications, which locate in a hidden directory, do not have the flag and executing it does not generate an alarm. Anyone, who knows where the legitimate application

is, can execute it without triggering an alarm. Adversaries can do so, if they figure out the hidden path. In this case, it is meaningless to check the flag because they can use the hidden interface by using the modified library.

Therefore, flag x is set to a file regardless of the interface. If the flag is set, an alarm is generated whenever it is executed no matter whom it is executed by. Legitimate applications do not have the flag and can be executed without an alarm by anyone who knows their location.

Outside attackers do not have permission to use the hidden interface, which means they do not know the hidden path. Inside attackers know the PhantomFS-v2 configuration, but cannot access the hidden interface. They will be detected if they run any flagged utility which does not locate in the hidden path. Impersonation attackers have permission to use one of the hidden interfaces. However, they do not know which utility is flagged. Since they know running a flagged utility triggers an alarm, they will be careful to run a utility unless they figure out flag configurations.

B. ANALYSIS OF ATTACK VECTORS

1) COMMAND HISTORY

The command history file (e.g. `.bash_history`) and the utility showing the command history (e.g. `history`) should be flagged so that they can be accessed only by legitimate users. In other words, if outside and inside attackers try to access them, an alarm is generated. If impersonation attackers try to access the command history of other users who use different hidden interfaces, an alarm is generated. The impersonation attackers may access their own history, but not that of others who use different interfaces.

2) UTILITIES BASED ON `PROC`

There are utilities (e.g. `ps`) that show the absolute path of applications. They are working based on the `proc` file system. We may disable or change the path of the `proc` file system, but it would cause many utilities not to work. Our approach is to replace the string of the hidden path to a special string (e.g. `*******`). We modify the source code of the `proc` file system to find the hidden path and replace it with the special string. To make it easier to find the hidden path, we enforce a naming rule for the hidden directory. If the hidden directory begins with "`PhantomFS_HiddenDir`", the whole path including the directory name is replaced by the special string by the `proc` file system. For example, flag h is set to `/home/admin/PhantomFS_HiddenDir_123qweASD`, the whole string is replaced by `********`.

3) REVERSE ENGINEERING

The hidden interface should be implemented in various ways to make it harder for adversaries to figure out how the hidden interface works by reverse engineering. The original paper implements the hidden interface by modifying the `read` system call. It distinguishes whether the `read` system call is used as a hidden interface or a regular interface by checking the signature. It is possible to implement multiple hidden interfaces by using multiple signatures, but it is weak to defend against reverse engineering. If adversaries manage to figure out how the `read` system call works, they can figure out all hidden interfaces.

Alternatively, we can use other system calls that take a user-space pointer as a parameter. The type of the pointer does not matter because we will use it in a different way. Once the system call is called, it reads the first few bytes from the pointer to check the signature. If the signature matches, it reads the rest from the pointer, which is the data structure to process the hidden interface. If the signature does not match, the call is processed in a regular way. There are many candidate system calls that can be used to implement the hidden interface in this way.

4) CONTROL FLOW HIJACKING

By exploiting vulnerabilities (e.g. memory corruption and command injection), adversaries may hijack the control flow of a legitimate application which uses a hidden interface. If it happens, they can access resources that the hidden interface is permitted to. This is the same situation with where impersonation attackers have the permission to use a hidden interface. However, if the adversaries access forbidden files while hijacking the control flow, PhantomFS-v2 can detect it. Even if it cannot do it at the exploitation stage (where vulnerabilities are exploited to hijack the control flow), it can detect it at the payload stage (where the malicious code is executed), if the malicious code accesses forbidden files.

For example, let us suppose adversaries want to access a specific target file which they are not allowed to, by hijacking the control flow of a legitimate application. To do this, they should escalate their privilege horizontally or vertically. Even if they succeed in privilege escalation, it does not necessarily mean they take the appropriate hidden interface to access their target file. They should find and take the hidden interface that is allowed to access the file they want. Otherwise, adversaries are detected if they access files that they are not permitted to.

5) MALWARE INSTALLATION

If adversaries cannot figure out which utility is safe to run, they may try to install their own. Malware installation often requires information gathering about the application, transferring or creating files of the malicious code, and installing the code. All of these steps should be done by running utilities on the host. Therefore, flagging utilities can prevent malware installation by outside attackers and inside attackers who cannot use the hidden interface.

Impersonation attackers, however, can access one of hidden interfaces, which means that they know where the modified library is. Thus, they can install malware and link it with the modified library so that it can access the hidden interface. In fact, most of highly secured servers do not have any development tools such as compilers (e.g. `gcc`) and script interpreters (e.g. `python`). However, casual servers

or internal servers used for testing or development may not follow the strict security policy and may have development tools. If impersonation attackers have permission to use the development tools, they may succeed in installing their malware which uses the hidden interface. This case is same with the control flow hijacking. Even if they manage to install malware, it is detected if it accesses forbidden files.

6) FILE ACCESS IN A NON-STANDARD WAY

The standard way to read and write a file is calling corresponding `read` and `write` system calls. However, it is possible to map the file to the memory and access the file as if accessing the memory. It can be done by calling the `mmap` system call. Thus, PhantomFS-v2 checks the flags when it is called.

Instead of overwriting a file, which triggers an alarm if flag w is set, adversaries may delete the file and create a new file with the same name. It is also possible to delete the target file and rename an existing file to the target file. In PhantomFS-v2, the flags are checked for unlinking and renaming files.

Adversaries may try to access files bypassing the file system. For example, they may use `debugfs` to access a file. Since it requires running existing utilities or install new executables, above-mentioned defense mechanisms can be used to mitigate it.

C. LIMITATIONS

Through the analysis of attack vectors, we believe that PhantomFS-v2 can successfully thwart attempts of outside and inside attackers who do not have permission to use a hidden interface. If impersonation attackers try to access files which are not permitted, PhantomFS-v2 can detect it. However, it cannot handle the following cases; (1) if impersonation attackers access files which are allowed via its hidden interface, and (2) if adversaries hijack the control flow of a legitimate application and access files which are allowed via the hidden interface of the legitimate application. To limit the impact of these cases, the least permission should be granted to each hidden interface.

V. EXAMPLE

In this section, we illustrate how the defense strategy works with an example.

A. SETUP

Let us suppose a web server as an example. As summarized in Table 3, it has three accounts. The `admin` account is for the system administrator, and the `webmaster` account is for the administrator of the website that the web server services. They use hidden interfaces, H_{admin} and H_{web}, respectively. As explained before, to use the hidden interface, they should run the script to change the default search path to the hidden path. Their hidden paths are denoted as P_{admin} and P_{web}, for H_{admin} and H_{web}, respectively. The `nobody` account is for the web server. The web server is continuously running with the modified library that uses the hidden interface H_{nobody}.

TABLE 3. User accounts of the example system.

Account	Role	Interface
`admin`	Administrator	H_{admin}
`webmaster`	Web master	H_{web}
`nobody`	Web server	H_{nobody}

Besides the web server, the system offers executables as shown in Table 4. According to the previous work [36], they are among the most executed commands by attackers. `ls` is a utility retrieving a list of files in a directory, and `vi` is a basic text editor. They typically locate in `/bin`. Thus, those in `/bin` are set with flag x, which triggers an alarm if anyone runs them. Legitimate users should use another version of them in their own hidden path, which does not trigger an alarm. Those in `/bin` are used to allure adversaries. The `ps` command shows the status of processes, which is allowed only for the administrator. It locates only in the hidden path of the administrator (P_{admin}).

TABLE 4. Executables of the example system.

Executable	Location	Flag
`ls`	`/bin`	x
`ls`	P_{admin}	
`ls`	P_{web}	
`vi`	`/bin`	x
`vi`	P_{admin}	
`vi`	P_{web}	
`ps`	P_{admin}	

In the example system, there are two data files that need protection. One is `index.html` which is serviced by the web server. As shown in Table 5, flag w is set to it for the all interfaces except for H_{web}. Thus, if an adversary, who uses the interface other than H_{web}, tries to modify it, an alarm is generated. Flag w is also set for H_{nobody} because the server only reads it. Even the administrator cannot access it because w flags are set for H_{admin}. Since the administrator is not supposed to manage the web site, modification to `index.html` is not allowed. The administrator may have privilege to access it (allowed by the conventional access control mechanism), but PhantomFS-v2 prevents the access. It plays as an additional barrier when the system is compromised. The other file under protection is `secret.txt` which is maintained by the administrator. Thus, flags rw are set for the regular interface and hidden interfaces except for H_{admin}.

TABLE 5. Flag configuration of the example system.

File	Regular	H_{admin}	H_{web}	H_{nobody}
`index.html`	w	w	-	w
`secret.txt`	rw	-	rw	rw

B. OUTSIDE ATTACKERS

Outside attackers may succeed in launching a shell by exploiting the vulnerability of the web server. The shell inherits the privilege of the `nobody` account and uses H_{nobody}. Instead, they may inject a malicious code to the web server by exploiting its vulnerability. In this case, the malicious code can use H_{nobody} because the web server is running by the `nobody` account. Both are similar to a situation where the outside attackers impersonate `nobody`. Through H_{nobody}, however, they cannot modify `index.html` nor access `secret.txt`.

C. IMPERSONATION ATTACKERS

Impersonation attackers may succeed in stealing credentials of a legitimate user, including the location of hidden path. In this example, let us suppose they steal those of `webmaster` and can use H_{web}, which means they know P_{web}. However, they do not know which executable is safe to run because they do not know the configuration. What they know for sure is that they can use `ls` and `vi` in P_{web}.

Impersonation attackers may modify `index.html` because they know all credentials of the legitimate web master. This case is similar with that of traitors, and cannot be handled by PhantomFS-v2. However, if they try to access `secret.txt`, PhantomFS-v2 can detect it.

They cannot access it with the privilege of `webmaster`. They need to escalate their privilege. Even if they succeed in privilege escalation, they are still unable to access `secret.txt` unless they figure out H_{admin}. Thus, PhantomFS-v2 plays an additional security barrier. In addition, PhantomFS-v2 limits resources available to impersonation attackers. Since they do not know which executable is safe to run and which file is safe to access, they can only use what are available in their own hidden path. Furthermore, the attempt of privilege escalation may trigger an alarm if flagged files are accessed while the attempt is being made. Therefore, they only have very limited attack vectors for privilege escalation and acquiring the hidden interface.

D. INSIDE ATTACKERS

Though inside attackers know details of PhantomFS-v2, they do not have the permission to access the web server. Thus, they are in a similar situation with outside attackers. What they can do is to attack the web server by exploiting its vulnerability. However, as discussed above, they cannot succeed in accessing protected resources unless they figure out the hidden interface.

VI. ADDITIONAL FEATURES

In this section, we present how we implement the additional features to realize the defense strategy discussed in Section IV.

A. FLAG CHECKING IN MORE SYSTEM CALLS

Table 6 shows which flags are checked in which system calls. We check flags in all system calls that can be potentially used to access files.

TABLE 6. Flags checked in system calls.

System call	Flags
`read`	r
`write`	w
`getdents`	h, r
`pread`	r
`pwrite`	w
`execve`	x
`mmap`	r, w, x
`memfd_create`	r, w
`rename`	r
`unlink`	r

In the case of `pread` and `pwrite`, they are system calls that read or write a file counting bytes from the offset position of the file. Adversaries may read or write files without triggering an alarm in the original PhantomFS. Therefore, in PhantomFS-v2, the code to check flags r and w is added in `pread` and `pwrite`, respectively.

As discussed in Section IV, we add a new flag, x, to detect running executables. The `execve` system call in the `exec` family is modified so that an alarm will be triggered when a file with flag x is executed. With PhantomFS-v2, it is possible to detect if an unauthorized user runs a flagged executable file.

Adversaries can use the `mmap` system call to read, write, and execute files without the conventional `read`, `write`, and `execve` system calls. In this way, the file with the flag can be accessed without alarm. In PhantomFS-v2, we modify the `mmap` system call to prevent it. The `mmap` system call takes `prot` argument which indicates the mapped file is to be read, written or executed. Therefore, `mmap` system call is modified to check the `prot` argument in PhantomFS-v2 to check the access to the file according to each flag. In the case of reading, if `prot_read` flag of `prot` is set, an alarm is triggered for a file with the r flag. In the same way, by checking `prot_write` for flag w and `prot_exec` for flag x, it is possible to detect access to a file using `mmap` system call rather than the usual system calls. The legitimate user can use the `mmap` system call implemented in the hidden interface through a library modified to use the legitimate hidden interface.

And we modify the `memfd_create` system call. This system call can also access files through memory mapping like `mmap`. So we add r flag and w flag check code to detect them.

Instead of directly modifying the contents of a file, adversaries may overwrite the file by `cp` and `mv`. The use of these commands can also be detected with PhantomFS-v2. Since the `cp` command uses the `write` system call, it can be detected if the w flag is set. This can also be detected by the original PhantomFS, but in the case of `mv` command, it calls another system call `rename`. Therefore, in PhantomFS-v2, we modify the `rename` system call to detect the overwriting of an existing file using the `mv` command if the r flag is set on the file. The legitimate user can use the `rename` system

call implemented in the hidden interface through the modified library.

Finally, the original PhantomFS was unable to detect file deletions. However, since an attacker can delete the file, we modify the `unlink` system call to detect the deletion of the file. It is possible to detect when an attacker deletes a file with flag *r*.

B. MULTIPLE HIDDEN INTERFACES

To implement multiple hidden interfaces, we assign different signature to each interface. However, if we use one system call for multiple interfaces, it may be vulnerable to reverse engineering, as discussed before. Thus, we use multiple existing system calls to support multiple hidden interfaces. 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 original PhantomFS implements the hidden interface using the `read` system call. For multiple hidden interfaces, we implement them using other system calls that also use a pointer to the user space. In this section, we illustrate them with `read`, `write` and `pread` system calls.

The process of using the hidden interface is basically same with the original PhantomFS: the application should allocate a buffer and write the signature, request type, and parameter in the buffer, and then call the system call, which is used for the hidden interface. PhantomFS-v2 supports 12 types of requests: (1) change flags, (2) read flags, (3) read, (4) write, (5) pread, (6) read a directory, (7) pwrite, (8) execve, (9) mmap, (10) memfd_create, (11) rename and (12) unlink. The first five types are supported by the original PhantomFS, and the remaining types are added in v2.

If the signature matches for each system call in kernel, it means it is called as a hidden interface. To implement multiple hidden interfaces, an interface index is given, when the signature matches, to identify which hidden interface is being used. For details, the pseudocode of the multiple hidden interfaces is given in Algorithm 2.

For example, the `read` system call is used for one of the hidden interfaces. When it is called, the signature is checked first. If it matches, `IF_READ` is assign to the `IF_INDEX` number and the `hidden_interface` function is called to process the request. `IF_INDEX` indicates which hidden interface is called. The actual value of `IF_READ`, `IF_WRITE` and `IF_PREAD` is randomized to make reverse engineering hard. If the signature does not match, it means the `read` system call is called as a regular interface. Then, `IF_REG` is assigned to the `IF_INDEX` number. If its flag *r* is set, a report is sent to the administrator. This is handled by `send_report` function. It sends a report to the administrator if any of flags (`FLAGS`) is set to the file for the interface (`IF_INDEX`). Finally, the original `read` system call is called. The `write` and `pread` system calls work in the same way.

Function `hidden_interface` is used to process the requests of the hidden interface with the `IF_INDEX` number. If the request type is changing flags, the `IF_INDEX` and flags are stored accordingly. If the request type is reading flags, flag information of the file for `IF_INDEX` is returned. If the request type is `read`, `write` or `pread`, the flags are checked by calling `send_report`. Then the original system call is called. Finally, the signature is erased to prevent the hidden interface is triggered unintentionally.

Other system calls shown in Table 6 are also modified to check the flags. In `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. In system calls `pwrite`, `execve`, `mmap`, and `rename`, flags are checked. The checked flags vary with

Algorithm 2 Pseudocode of a Hidden Interface Implementation With Examples of the Read, Write and Pread System Calls

```
1:  procedure READ
2:      if the signature matches then
3:          call hidden_interface function(IF_READ)
4:          return
5:      call send_report(IF_REG, r)
6:      call the original read system call
7:  procedure WRITE
8:      if the signature matches then
9:          call hidden_interface function(IF_WRITE)
10:         return
11:     call send_report(IF_REG, w)
12:     call the original write system call
13: procedure PREAD
14:     if the signature matches then
15:         call hidden_interface function(IF_PREAD)
16:         return
17:     call send_report(IF_REG, r)
18:     call the original pread system call
19: procedure PWRITE
20:     call send_report(IF_REG, w)
21:     call the original pwrite system call
22: procedure SEND_REPORT (IF_INDEX, FLAGS)
23:     if FLAGS are set to the file for IF_INDEX then
24:         send a report
25: procedure HIDDEN_INTERFACE(IF_INDEX)
26:     if the request type is read then
27:         call send_report(IF_INDEX, r)
28:         call the original read system call
29:     if the request type is write then
30:         call send_report(IF_INDEX, w)
31:         call the original write system call
32:     if the request type is pread then
33:         call send_report(IF_INDEX, r)
34:         call the original pread system call
        ... process other request types ...
35:     erase the signature
```

system calls as shown in Table 6. Since they are not used as a hidden interface, however, the signature is not checked. In the `hidden_interface` function, if the request type is `getdents`, the original `getdents` system call is called, and then the files with flag `h` for `IF_INDEX` are removed from the list. For other system calls, their corresponding flags are checked by calling `IF_INDEX` and their original system calls are called.

The data structure storing the flags is maintained individually for each hidden interface, as illustrated in Figure 3. Flags are stored per file and per interface. The stored flags of a file are checked when the file is accessed through *other* interfaces including the regular interface. Let us consider the example of Figure 3. Let us suppose user 1 is using hidden interface 1. User 1 sets flags *rx* to file 1. This means if file 1 is read or executed through *other* interfaces, an alarm is triggered. At the same time, *w* flags is set to the same file through hidden interface 2. Thus, if user 1 tries to overwrite file 1, an alarm is triggered. Since the data structure is maintained separately, user 1 cannot modify nor read flag *w* of hidden interface 2.

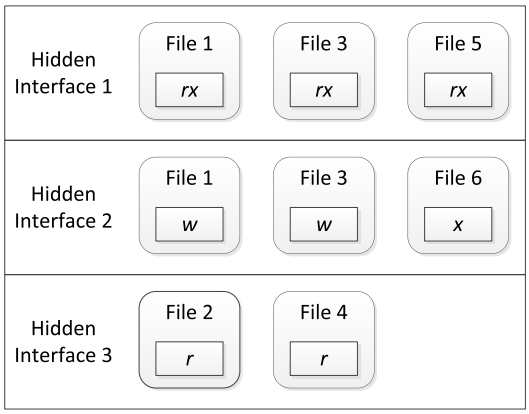


FIGURE 3. Illustration of the data structure maintained to store flags. Flags of a file are checked when the file is accessed through *other* interfaces.

We modify the `glibc` code to use the hidden interface. To process 12 types of requests, we modify their wrapper functions. In these functions, we create signature, request type and parameter in a buffer and call the corresponding system call, as illustrated in Figure 4. The hidden interface implementation by using the `read` system call is illustrated in Figure 4. When the application calls the `write` wrapper function, `glibc`'s `write` wrapper function creates a buffer that makes a write request to PhantomFS-v2 for using the hidden interface and calls the `read` system call.

VII. EXPERIMENTS

In this section, we demonstrate that PhantomFS-v2 can detect adversaries who are aware of PhantomFS-v2, which cannot be detected by the original PhantomFS. We demonstrate it by penetration testing. We also measure the performance overhead caused by PhantomFS-v2 and show it is negligible.

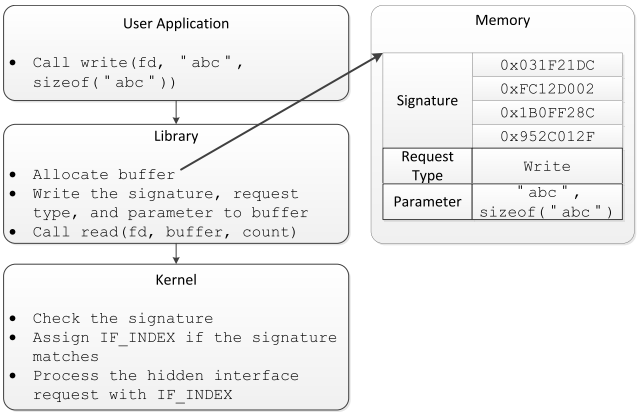


FIGURE 4. Hidden interface implementation by using the `read` system call.

TABLE 7. Machine specification used for experiments.

Host machine	Processor	Intel i7
	Memory	32 GB
	Disk	512 GB SSD
	Operating system	Windows 10
Virtual machine	Virtualizer	VirtualBox 5.2.26
	Memory	4 GB
	Disk	50 GB
	Operating system	Ubuntu Server 18.04
	Kernel	Linux 5.0.7

A. EXPERIMENTAL SETUP

We use a similar setup with the example given in Section V. We install Ubuntu Server 18.04 as a virtual machine on a host machine. Their spec is given in Table 8. We install GoAhead [37] as a web server running on the virtual machine. This server is one of the most popular embedded web server which is efficient for developers to host embedded web applications. We utilize CVE-2017-17562 [38], a severe vulnerability which allows remote code execution for GoAhead before 3.6.5, as the capability of some type of attackers for our scenarios. This vulnerability enables the attackers to obtain the privilege which allows for execution of arbitrary

TABLE 8. Comparison against existing techniques. [Detection: O means detected, X means not detected, and △ means potentially detected (but not guaranteed)] [False alarm: O means false alarm may be generated and X means no false alarm is generated].

Category	Technique	Scenario 1	Scenario 2
Detection	PhantomFS-v2	O	O
	PhantomFS	X	X
	File integrity checking	O	X
	Honeyfile	△	△
False alarm	PhantomFS-v2	X	X
	PhantomFS	X	X
	File integrity checking	O	X
	Honeyfile	O	O
Overhead	PhantomFS-v2	Low	
	PhantomFS	Low	
	File integrity checking	High	
	Honeyfile	Low	

commands remotely through the GoAhead web server. The open ports are 22 and 80, which are for the secure shell (SSH) and the web server, respectively. No development tools are installed.

We created three accounts as like in Table 3. The hidden path for the administrator (P_{admin}) is `/home/admin/PhantomFS_HiddenDir_q1W2e3` and that for the web master (P_{web}) is `/home/web/PhantomFS_HiddenDir_789uioJKL`. The modified library and the script to change the search path are placed in the hidden path. Flag h is set to the hidden directory for all interfaces except for the corresponding hidden interface. Specifically, h is set to P_{admin} for the regular interface, H_{web} and H_{nobody}; and to P_{web} for the regular interface, H_{admin}, and H_{nobody}.

All executables in `/bin`, `/sbin`, `/usr/bin`, and `/usr/sbin` are flagged with x. They are copied to P_{admin} without the flag so that the administrator can use them. Those in P_{admin} are linked with the modified library by changing the path of the library. For the web master, only essential utilities are copied to P_{web}. They are `ls` and `vi` in this penetration testing.

B. PENETRATION TESTING

In this subsection, we validate the security of PhantomFS-v2 by penetration testing.

1) ATTACK GOALS

We place `index.html` and `secret.txt` as the targets of attacks. Flags are set to them as shown in Table 5.

The goal of outside attackers and inside attackers is to modify `index.html` without triggering an alarm. We assume the impersonation attackers succeed in stealing the credentials of `webmaster`, which include its login password and hidden path. The goal of impersonation attackers is to read `secret.txt` without being detected by PhantomFS-v2.

2) COMPARISON WITH ALTERNATIVES

We compare the detection capability and false alarm of PhantomFS-v2 against the original PhantomFS, file integrity checking, and honeyfile [8]. The file integrity checking technique is a very well-known technique that detects unauthorized modification to files. It maintains the integrity metric (typically hash) of files, and detects modification by comparing the integrity metric. The honeyfile technique is a file-based deception technology that detects intrusion when adversaries access decoy files.

For comparison, we use the following two attack scenarios. There could be other scenarios to achieve the two attack goals. We test other scenarios at the end of this subsection.

Scenario 1: The outside and inside attackers are enabled to obtain code execution privilege on the target host by exploiting the vulnerability of the web server.

- The attackers exploit a vulnerability in the web server to obtain remote command execution.
- Despite of knowing existence of PhantomFS-v2, the attackers inadvertently modify `index.html` with `/bin/vi` executable.

Scenario 2: Impersonation attackers have the credentials of the `webmaster` account and can access hidden interface H_{web}, which enables they can make use of executables from P_{web}. However, they are not conscious of exact flag configurations for executables, which means they are unlikely execute any binaries other than `vi` and `ls` in P_{web}.

- The impersonation attacker gets access to H_{web} interface.
- The attacker reads `secret.txt` by using `vi` in P_{web}.

For the original PhantomFS, we set flag w to `index.html` and r to `secret.txt`. All users (administrator, webmaster, and web server) use the hidden interface. Recall that there is only one hidden interface in the original PhantomFS.

For file integrity checking, we calculate and keep the hash values of `index.html` and `secret.txt` a priori. After executing the attack scenarios, we calculate their hash values again and compare them to the stored values.

For the honeyfile, we cannot use the real files as decoy files. Thus, we place fake decoy files with confusing file names. We place `index.htm` as a decoy file for the first scenario, and `confidential.txt` as a decoy for the second scenario.

PhantomFS-v2 successfully detects both scenarios, whereas the original PhantomFS cannot. This is because there is only one hidden interface in the original PhantomFS. Since the outside and inside attackers gain access to the web server, they can use the hidden interface and access `index.html` without triggering the alarm. In a similar vein, the impersonation attackers can access `secret.txt` without being detected. If we employ file integrity checking, file modification (scenario 1) can be detected, but file read (scenario 2) cannot. When decoy files are placed, the success of detection depends on whether the attackers are deceived or not. If they are confused and access decoy files, they are detected. However, if they do not access decoy files but only the target files, the honeyfile technique cannot detect the unauthorized file access.

Both PhantomFS-v2 and original PhantomFS do not generate false alarms because they use the hidden interface to prevent legitimate users from triggering alarms accidentally. File integrity checking may generate false alarms if the legitimate user updates target files. The false alarm is one of the biggest concern of the honeyfile because the legitimate users may access the decoy files by mistake and the legitimate background process (e.g. indexing and file searching) may access them as well.

The performance overhead of PhantomFS-v2, original PhantomFS and honeyfile is lower than that of the file integrity checking. This is because the file integrity checking technique checks the integrity by reading all target files and computing the integrity metric whereas other techniques only detect access to target files.

3) VALIDATION OF DEFENSE STRATEGY

Once attackers gain access to a host by exploiting a vulnerability (scenario 1) or impersonating a legitimate user (scenario 2), they may try to find the hidden interface or the hidden path, or to evade PhantomFS-v2. We discuss possible attack vectors and develop defense mechanisms as summarized in Table 2. By penetration testing, we validate the defense mechanisms.

The following scenarios are used to validate the defense mechanisms. They are of the impersonation attackers who have stolen the credentials of `webmaster` and try to access `secret.txt`.

Scenario 3 (Command History): Impersonation attackers are aware of the fact that they can access `secret.txt` without being detected if they can figure out P_{admin}. They may try to do this by looking at the command history of the administrator.

- The administrator set flag *r* to `.bash_history`.
- The impersonation attacker logins as `webmaster`.
- The attacker reads `.bash_history` of `admin` by using `vi` in P_{web}.

Even though the attacker uses `vi` in P_{web}, the attacker is detected because `.bash_history` has flag *r*. The attacker may also try to retrieve the command history by using `lastcomm`, but it does not reveal the command line argument nor the absolute path of the command.

Scenario 4 (Utilities Based on `proc`): Impersonation attackers may try to find the absolute path of legitimate applications located in P_{admin}..

- The impersonation attacker logins as `webmaster`.
- The attacker reads `/proc/1522/cmdline` using `vi` in P_{web}.

In this scenario, PID 1522 is of the legitimate application located in P_{admin}. To find its absolute path, the attacker reads `/proc/1522/cmdline`, but the path is replaced with ********, which prevents the attacker from finding the absolute path.

Scenario 5 (Reverse Engineering): The attackers can figure out how PhantomFS-v2 works by analyzing binaries related to hidden interface. They know which functions are called to handle hidden interface requests and find some signatures used to identify if the request is correct. Knowing hidden interface signatures, the attackers expect that they can modify flag configurations on any files by calling file related system calls with fabricated user-space pointer as an argument.

- The impersonation attacker logins as `webmaster`.
- The attacker crafts fake memory pointer with hidden interface signature values to request read or write on `secret.txt`.

Since the attacker requests file related system calls by forging memory pointer as an argument with all hidden interface signature values but an interface identifier which is generated randomly at run-time, PhantomFS-v2 alerts an alarm on account of interface inconsistency.

Scenario 6 (Control flow hijacking): This scenario is already covered by scenario 1.

Scenario 7 (Malware installation): The attackers may try to bring malware into the host. It can be detected when the file is transferred, when the malware is compiled or installed, or when it is executed. In this scenario, it is detected at the moment when it is transferred.

- The impersonation attacker logins as `webmaster`.
- Since the attacker cannot bring a file by using the utilities available in P_{web}, he inadvertently runs `wget` to copy a file from a remote host.

Since *x* flag is set to `wget`, the attacker is detected when it is executed.

Scenario 8 (File access in a non-standard way): The attackers may try to destroy `secret.txt` instead of reading it.

- The impersonation attacker logins as `webmaster`.
- The attacker creates a file whose name is `secret.txt` in his own local directory.
- The attacker copies it to the administrator's directory by using the `cp` command to overwrite the target `secret.txt` file.

The attacker is detected not only by executing `cp` which has flag *x*, and also by the `unlink` system call being called, which checks flag *r* of `secret.txt`.

C. PERFORMANCE EVALUATION

We measure the performance overhead of 10 system calls we modified. We create a script that calls each system call, and record the response time 100 times. Figure 5 shows the average of 100 experiments by a thick histogram bar, and the maximum and minimum by a narrow error bar. We compare the response time of the unmodified file system without PhantomFS-v2 ('Unmodified'), the response time of PhantomFS-v2 using the regular interface ('Regular'), and that of PhantomFS-v2 using the hidden interface through the `read` system call ('Hidden').

The overhead of the `read` system call occurs when it is used as a regular interface ('Regular') because the signature is checked and the flag is checked. As shown in Figure 5(a), we measure the response time of Regular varying the size of requests. When the request size is small, the relative overhead is high. When the request size is 4K, the response time is increased by 103.00 % on average. This may look excessive, but the absolute increase is only 4.12 ms. When the request size is 4M, the response time is increased by 19.55 % on average.

When the `read` system call is called as a hidden interface, the overhead occurs by checking the signature, processing the request and checking the flag. When the request size is 4K, it is 85.00% and when the size is 4M, it is 21.06%.

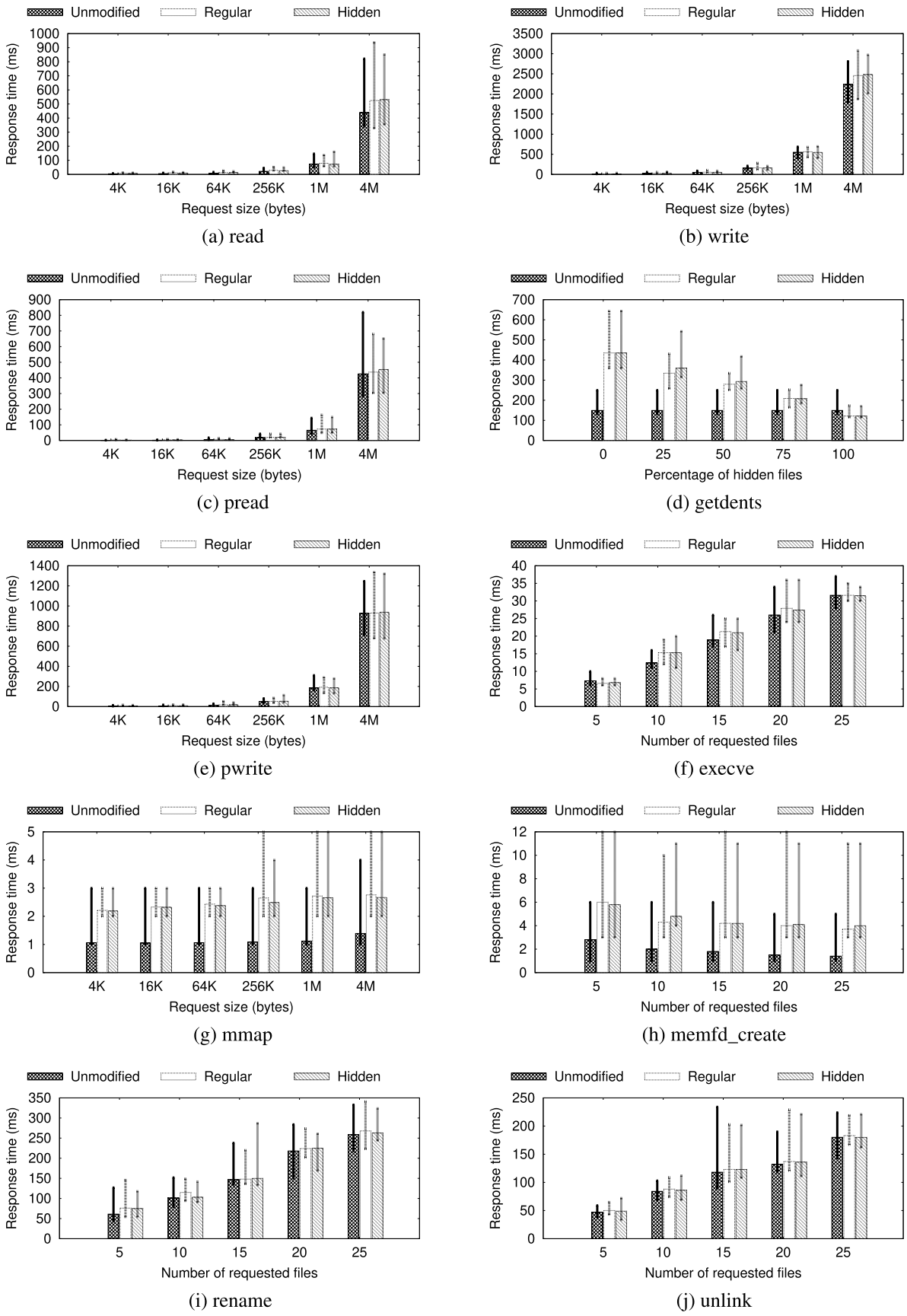


FIGURE 5. Comparison of response time in system calls. The response time is measured through the system call of the unmodified kernel ('Unmodified'), the modified regular interface of PhantomFS-v2 ('Regular'), and the hidden interface of PhantomFS-v2 ('Hidden').

It should be noted that the overhead experienced by the user is much lower than this experimental result. The experimental result is the overhead of one system call, but the system call is only a part of a utility. Thus, the ratio of the increased response time of a utility must be lower than that of a system call.

To demonstrate this, we compare the throughput of the `dd` utility in Figure 6. The utility reports the throughput of accessing a disk. Note that the y-axis is throughput, where the higher is the better. As shown in this figure, there is no significant difference in the throughput of Unmodified, Regular, and Hidden, because the `read` system call is only a part of the `dd` utility. The overhead of Regular is 3.35% – 8.21% and that of Hidden is up to 0.58%. In case of Hidden, the throughput is sometimes higher than Unmodified. It means that the slight increase of the response time in the `read` system call does not have significant impact on the overall performance of the `dd` utility.

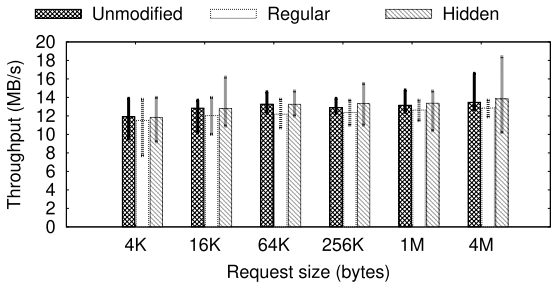


FIGURE 6. Comparison of throughput measured by `dd`.

For system calls `write`, `pread`, and `pwrite`, we observe a similar trend. The overhead is 2.40% – 24.64% (Regular) and 1.23% – 19.20% (Hidden) in case of the `write` system call. It is 2.66% – 199.05% (Regular) and 6.67% – 111.32% (Hidden) for `pread`; and it is 0.25% – 91.89% (Regular) and 0.89% – 95.25% (Hidden) for `pwrite`.

In the `getdents` system call, the overhead is caused by an additional memory copy. The `getdents` system call gets a list of all files in that directory. Then files with *h* flag are removed from the list while being copied to another memory location, which incurs the additional memory copy overhead. Thus, the overhead decreases as the number of hidden files increases.

The figure 5(d) shows the response time of the `getdents` system call. We create 100 files and measure the response time when the 0%, 25%, 50%, 75%, and 100% of files have *h* flag. It can be seen that as the number of hidden files increases, the overhead decreases. When the percentage of hidden files is 100%, the response time of Regular and Hidden is even shorter than that of Unmodified. Except for this, the overhead is 40.01% – 191.80% (Regular) and 39.25% – 191.80% (Hidden).

The `execve` system call is used to execute a file. We measure its response time varying the number of requested files. The response time of Regular and Hidden is sometimes lower than that of Unmodified, but the difference is within the noise margin. In other cases, the overhead is 7.30% – 23.38% (Regular) and 5.38% – 23.38% (Hidden).

In the case of `mmap` system call, overhead occurs when checking the `prot` argument and flag. Once a file is mapped to the memory, it is accessed directly from the memory. Thus, the overhead occurs only when the file is mapped and it is constant regardless of request size, as shown in Figure 5(g). The overhead is 98.56% – 143.11% (Regular) and 92.08% – 128.44% (Hidden). However, the absolute value is only 1.15ms – 1.60ms (Regular) and 1.13ms – 1.54ms (Hidden).

The `memfd_create` system call is used when an anonymous file is created. Thus, we measure its overhead varying the number of requested files. The overhead is caused only by checking the flag. The overhead is 114.28% – 166.66% (Regular) and 107.14% – 185.71% (Hidden), but the absolute value is only 2.14ms – 2.66ms (Regular) and 2.07ms – 2.85ms (Hidden).

In `rename` and `unlink` system calls, the overhead occurs only for checking the flag. We measure the response time varying the number of requested files. The overhead is 0.68% – 24.59% (Regular) and 1.54% – 22.95% (Hidden) for `rename` and it is 1.66% – 6.38% (Regular) and 2.38% – 4.25% (Hidden) for `unlink`.

We also compare the CPU time, measured by `iozone` utility. We compare the CPU time of `read` and `write` system calls because they are used to implement hidden interfaces. The results are shown in Figure 7. Since the CPU time is relatively very short compared to the total response time, we use larger file sizes than Figure 5.

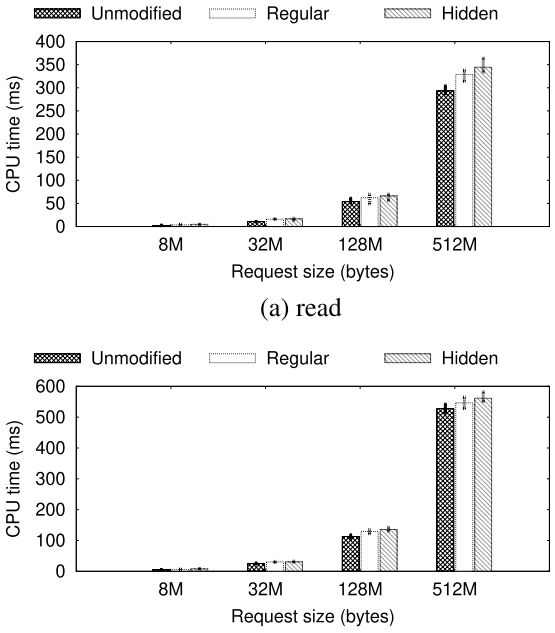


FIGURE 7. Comparison of CPU time of read and write system calls measured by `iozone`.

VIII. CONCLUSION

The file-based deception technology has been demonstrated to be effective in thwarting malicious users who have gained access to the host evading intrusion detection systems [9]. However, if adversaries become aware of the deception technology, the deception technology is unlikely to succeed in alluring adversaries. In this paper, we analyze the attack surface of the file-based deception technology and propose PhantomFS-v2 which offers countermeasures to intelligent adversaries who are aware of the file-based deception technology. By penetration testing, we demonstrate that even if adversaries are aware of PhantomFS-v2, they cannot figure out the hidden interface, and even if adversaries can access one of hidden interfaces, they cannot access other hidden interfaces. Though the response time of system calls increases a little, it does not have significant impact on the user's experience because the response time of a system call is only a part of user's applications. As a result, we expect that PhantomFS-v2 would be an effective countermeasure to those adversaries who are aware of the file-based deception technology. PhantomFS-v2 provides an additional protection mechanism for important files, which may not be fully protected by conventional intrusion detection systems.

ACKNOWLEDGMENT

(Jione Choi and Hwiwon Lee are co-first authors.)

REFERENCES

[1] L. Spitzner, "The honeynet project: Trapping the hackers," *IEEE Secur. Privacy*, vol. 1, no. 2, pp. 15–23, Mar. 2003.

[2] Wikipedia. *Honeypot (Computing)*. Accessed: 2020. [Online]. Available: https://en.wikipedia.org/wiki/Honeypot_(computing)

[3] 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 honeytoken based framework," in *Proc. 9th IFIP Int. Conf. New Technol., Mobility Secur. (NTMS)*, Feb. 2018, pp. 1–4.

[4] M. Bercovitch, M. Renford, L. Hasson, A. Shabtai, L. Rokach, and Y. Elovici, "HoneyGen: An automated honeytokens generator," in *Proc. IEEE Int. Conf. Intell. Secur. Informat.*, Jul. 2011, pp. 131–136.

[5] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proc. 2013 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 145–160.

[6] 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.*, 2014, pp. 942–953.

[7] D. Fraunholz, S. Duque Anton, C. Lipps, D. Reti, D. Krohmer, F. Pohl, M. Tammen, and H. Dieter Schotten, "Demystifying deception technology:A survey," 2018, *arXiv:1804.06196*. [Online]. Available: http://arxiv.org/abs/1804.06196

[8] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: Deceptive files for intrusion detection," in *Proc. from 5th Annu. IEEE SMC Inf. Assurance Workshop*, Jun. 2004, pp. 116–122.

[9] J. Lee, J. Choi, G. Lee, S.-W. Shim, and T. Kim, "PhantomFS: file-based deception technology for thwarting malicious users," *IEEE Access*, vol. 8, pp. 32203–32214, 2020.

[10] B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo, "Baiting inside attackers using decoy documents," in *Security and Privacy in Communication Networks*. Berlin, Germany: Springer, 2009, pp. 51–70.

[11] 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 Digital Forensic (CyberSec)*, Mar. 2013.

[12] M. Lazarov, J. Onaolapo, and G. Stringhini, "Honey sheets: What happens to leaked google spreadsheets?" in *Proc. 9th Workshop Cyber Secur. Experimentation Test(CSET)*, 2016, pp. 1–8.

[13] C. Moore, "Detecting ransomware with honeypot techniques," in *Proc. Cybersecurity Cyberforensics Conf. (CCC)*, Aug. 2016, pp. 77–81.

[14] R. Zhuang, S. A. DeLoach, and X. Ou, "Towards a theory of moving target defense," in *Proc. 1st ACM Workshop Moving Target Defense (MTD)*, 2014, pp. 31–40.

[15] Q. Jia, K. Sun, and A. Stavrou, "MOTAG: Moving target defense against Internet denial of service attacks," in *Proc. 22nd Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2013, pp. 1–9.

[16] T. E. Carroll, M. Crouse, E. W. Fulp, and K. S. Berenhaut, "Analysis of network address shuffling as a moving target defense," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2014, pp. 701–706.

[17] M. Christodorescu, M. Fredrikson, S. Jha, and J. Giffin, *End-to-End Software Diversification of Internet Services*. New York, NY, USA: Springer, 2011, pp. 117–130.

[18] P. Kampanakis, H. Perros, and T. Beyene, "SDN-based solutions for moving target defense network protection," in *Proc. IEEE Int. Symp. World Wireless, Mobile Multimedia Netw.*, Jun. 2014, pp. 1–6.

[19] E. Al-Shaer, "Toward network configuration randomization for moving target defense," in *Moving Target Defense*. 2011, pp. 153–159.

[20] H. Okhravi *et al.*, "Creating a cyber moving target for critical infrastructure applications," in *Proc. 5th Int. Conf. Crit. Infrastruct. Protection (ICCIP)*, Hanover, NH, USA, Mar. 2011, pp. 107–123.

[21] E. Al-Shaer, Q. Duan, and J. Jafarian, "Random host mutation for moving target defense," in *Proc. 8th Int. ICST Conf. Secur. Privacy Commun. Netw.*, 2012, pp. 310–327.

[22] X. Jiang, H. J. Wangz, D. Xu, and Y. Wang, "Randsys: Thwarting code injection attacks with system service interface randomization," in *Proc. 26th IEEE Int. Symp. Reliable Distrib. Syst. (SRDS)*, Oct. 2007, pp. 209–218.

[23] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," Tech. Rep., Apr. 2009.

[24] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, *Compiler-Generated Software Diversity*. New York, NY, USA: Springer, 2011, pp. 77–98.

[25] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. 10th ACM Conf. Comput. Commun. Secur. (CCS)*, 2003, pp. 272–280.

[26] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The effectiveness of instruction set randomization," in *Proc. 14th Conf. USENIX Secur. Symp.*, vol. 14. New York, NY, USA: USENIX Association, 2005, p. 10.

[27] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Comput. Commun. Secur. (CCS)*, 2004, pp. 298–307.

[28] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL injection attacks," in *Applied Cryptography and Network Security*, M. Jakobsson, M. Yung, and J. Zhou, Eds. Berlin, Germany: Springer, 2004, pp. 292–302.

[29] B. Whitham, "Automating the generation of enticing text content for high-interaction honeyfiles," in *Proc. 50th Hawaii Int. Conf. Syst. Sci.*, 2017, pp. 1–10.

[30] 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. Thankachan, and N. C. Debnath, Eds. Berlin, Germany: Springer, 2011, pp. 117–122.

[31] 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, 2018.

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

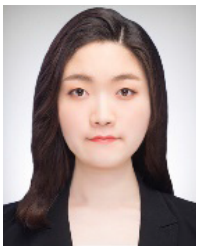
[33] S. N. Chari and P.-C. Cheng, "BlueBoX: A policy-driven, host-based intrusion detection system," *ACM Trans. Inf. Syst. Secur.*, vol. 6, no. 2, pp. 173–200, May 2003, doi: 10.1145/762476.762477.

[34] 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.

[35] S. Ganapathy, K. Kulothungan, S. Muthurajkumar, M. Vijayalakshmi, P. Yogesh, and A. Kannan, "Intelligent feature selection and classification techniques for intrusion detection in networks: A survey," *EURASIP J. Wireless Commun. Netw.*, vol. 2013, no. 1, p. 271, Dec. 2013.

[36] J. Briffaut, J.-F. Lalande, and C. Toinard, "Security and results of a large-scale high-interaction honeypot," *J. Comput.*, vol. 4, no. 5, pp. 395–404, May 2009.

[37] EMBEDTHIS. (2020). *Goahead: Simple, Secure Embedded Web Server*. Accessed: May 5, 2020. [Online]. Available: https://www.embedthis.com/goahead/
[38] NIST. (2017). *Goahead Web Server Before 3.6.5: HTTPd LD_PRELOAD Remote Code Execution*. Accessed: May 5, 2020. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2017-17562



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 for an AI company called Furiosa AI, as a Profiler Programmer for neural processing unit chip. Her research interests include Linux kernel, file systems, and security, such as intrusion detection and honey pot.

HWIWON LEE received the B.S. degree in cyber defense from Korea University, in 2017. He is currently pursuing the Integrated Ph.D. (by Program) degree in cybersecurity with Korea University. His research interests include embedded system security, software testing, and binary analysis. He won the DEFCON CTF, a worldwide hacking competition, in 2015 and 2018.

YOUNGGI PARK received the B.S. degree in cyber defense from Korea University, in 2018, where he is currently pursuing the M.S. degree in cybersecurity. His research interests include vulnerability analysis, automated root cause analysis, and reverse engineering.

HUY KANG KIM (Member, IEEE) received the B.S. and M.S. degrees in industrial engineering and the Ph.D. degree in industrial and systems engineering from the Korea Advanced Institute of Science and Technology (KAIST), in 1998, 2000, and 2009, respectively. He is a Serial Entrepreneur. He founded A3 Security Consulting, in 1999, and AI Spera, in 2017. He is currently a Professor with the School of Cybersecurity, Korea University.

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 with the Georgia Institute of Technology, in 2013. From 2003 to 2008, he worked at Samsung Electronics on electronic system level design of mobile system-on-chip. From 2014 to 2019, he was with the Department of Electrical and Computer Engineering, The University of Texas at San Antonio, as an Assistant Professor. He has been with the School of Cybersecurity, Korea University, since 2019. His research interests include secure design or hardware-assisted security of processor, non-volatile memory, storage, and dedicated hardware.

YOUNGJAE KIM (Member, IEEE) received the B.S. degree in computer science from Sogang University, Seoul, South Korea, in 2001, the M.S. degree in computer science from KAIST, in 2003, and the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA, in 2009. From 2009 to 2015, he worked as a Research and Development Staff Scientist with the U.S. Department of Energy's Oak Ridge National Laboratory. From 2015 to 2016, he was with Ajou University, as an Assistant Professor. He has been an Associate Professor with the Department of Computer Science and Engineering, Sogang University, since 2016. His research interests include operating systems, file and storage systems, parallel and distributed systems, and computer systems security.

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 Corporation, 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, 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 Jungang 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 at SK C&C software techniques team, in 2008. He has been working with LIG Nex1, South Korea, since 2010. His research interests include cyber operation systems, training systems, and cyber infra systems.