Ensuring Deception Consistency for FTP Services Hardened against Advanced Persistent Threats

Zhan Shu
Department of Computer Science
Binghamton University, State University of New York
zshu1@binghamton.edu

Guanhua Yan
Department of Computer Science
Binghamton University, State University of New York
ghyan@binghamton.edu

ABSTRACT

Recent security incidents such as the Target data breach and the Equifax hack suggest APTs (Advanced Persistent Threats) can significantly compromise the trustworthiness of cyber space. This work explores how to improve the effectiveness of cyber deception in hardening FTP (File Transfer Protocol) services against APTs. The main objective of our work is to ensure *deception consistency*: when the attackers are trapped, they can only make observations that are consistent with what they have seen already so that they cannot recognize the deceptive environment. To achieve deception consistency, we use logic constraints to characterize an attacker's best knowledge (either positive, negative, or uncertain). When migrating the attacker's FTP connection into a contained environment, we use these logic constraints to instantiate a new FTP file system that is guaranteed free of inconsistency. We performed deception experiments with student participants who just completed a computer security course. Following the design of Turing tests, we find that the participants' chances of recognizing deceptive environments are close to random guesses. Our experiments also confirm the importance of observation consistency in identifying deception.

KEYWORDS

Cyber deception, deception consistency, advanced persistent threat

ACM Reference Format:
Zhan Shu and Guanhua Yan. 2018. Ensuring Deception Consistency for FTP Services Hardened against Advanced Persistent Threats. In *5th ACM Workshop on Moving Target Defense (MTD '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3268966.3268971

1 INTRODUCTION

As evidenced by a number of recent high-profile security incidents such as the Target data breach [23] and the Equifax hack [1], advanced persistent threats (APTs) have posed unprecedented security challenges to Internet users. In the battle of mitigating the ever-growing APTs, cyber deception has recently regained popularity: multiple research firms have predicted that the cyber deception market will grow by more than 9 percent annually and reach more than one billion US dollars just in a few years [22]. By manipulating attackers' perceptions, cyber deception enables another layer of dynamic components to the attack surface of a system, which agrees well with the spirit of the moving target defense paradigm.

Although deception-based defenses such as honey pots have had a long history and played an important role in understanding various cyber threats in the past, it is still difficult, if not impossible, to reason about the effects of cyber deception on the adversary in practice. Particularly, when deceptive techniques are deployed to thwart APT attacks, they need to be planned carefully because an experienced APT attacker can easily recognize "bad lies." For many deception-based methods such as decoy and disinformation, their utility hinges upon the attackers being unaware of their existences.

Against this backdrop, our work explores one important aspect of cyber deception, *deception consistency*, which is crucial to its success in APT mitigation. Deception consistency ensures that the observations made by an APT attacker (or his malware surrogate) after he is trapped into a deceptive environment should be consistent with what he has previously seen so that he cannot recognize the deceptive environment. To gain deep insights into the challenges of ensuring deception consistency in specific APT attack scenarios, we harden existing FTP services with cyber deception capabilities: we monitor APT attackers' attempts at exploiting common FTP service vulnerabilities (e.g., buffer overflow attacks) and migrate the attackers' FTP connections into a deceptive environment aimed at revealing their origins, intents, and capabilities.

For FTP services hardened with cyber deception against APTs, we explore how to ensure consistency of the attacker's observations after he is diverted to a deceptive FTP service. We develop efficient methods to ensure the observation consistency of the attacker while migrating his FTP connection into a deceptive environment. In a nutshell, our contributions are summarized as follows:

- We formulate the problem of deception consistency assurance rigorously and develop techniques to ensure that within the deceptive environments where the attackers are trapped, they can make only observations consistent with what they have seen before. In our methods, we represent attackers' best knowledge about the FTP file system, either positive, negative, or uncertain, as logic constraints maintained within a tree data structure. When migrating the attacker's FTP connection, these logic constraints are used to instantiate a new FTP file system in the deceptive environment.
- We instrument FTP services with APT detection capabilities, and develop methods based on process checkpointing and restoring functionalities to migrate attackers' FTP connections transparently into contained VMs where attackers' intents, capabilities and tactics can be further revealed.

MTD '18, October 15, 2018, Toronto, ON, Canada
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6003-6/18/10...$15.00
https://doi.org/10.1145/3268966.3268971

- To study the effectiveness of our methods, we perform experiments with 32 student participants who have finished a computer security course. Following the design of Turing tests, we find that the participants' capabilities of recognizing deceptive environments are close to random guesses. The experiments also demonstrate the crucial importance of observation consistency in identifying deception.

The remainder of the paper is organized as follows. Section 2 introduces related work and Section 3 the threat model considered in this work. Section 4 provides our algorithm for ensuring deception consistency in protecting FTP services and Section 5 some implementation details. Section 6 presents the evaluation results of our method. In Section 7 we draw concluding remarks.

2 RELATED WORK

Even extended with various security features [17, 19], FTP services have been targets of hackers due to their popularity in enterprise network environments. A variety of attacks against FTP services exist, such as bounce attacks, port stealing and sniffing. In this work, we focus on defending against APT attacks aimed at exploiting vulnerable FTP services either to gain their initial foothold or to move laterally inside an enterprise network.

The original concept of cyber deception dated back to the 1990s when deception was used to baffle an intruder on an Internet gateway for several months [15]. Honeypots and honeynets have been widely deployed to understand emerging cyber threats such as worms, botnets, and phishing attacks [7, 8, 16, 25, 26]. Cyber deception has also been used to mitigate threats such as insider attacks, external reconnaissance, and network eavesdropping [14, 27, 29–32]. Decoy unpatched software vulnerabilities are used to confuse the attackers and then trap them into contained environments where they can successfully exploit unpatched vulnerabilities [12]. Similar to our work, they also uses live process migration to achieve transparent trapping of attackers. Deception-based methods have been evaluated [18] in detecting attacks against web applications. Although inspired by these previous works, the main focus of our study is to ensure consistency of attackers' observations throughout the life cycles of their attacks.

Developing rigorous methods for deception consistency assurance in cyber defense operations agrees well with the recent trend in building a scientific foundation for cyber deception [20]. Deception consistency has been discussed in a few previous works [21, 24, 28]. Although they have sharply observed the importance of consistency for the success of cyber deception, most of their discussions are done at the conceptual level without providing specific solutions to achieving deception consistency. To the best knowledge, our work offers the first systematic approach to ensuring deception consistency in a specific cyber attack and defense scenario.

3 THREAT MODEL

In our threat model, we consider APT attackers targeting vulnerable FTP services. The attackers may use anonymous FTP accounts, or stolen credentials of legitimate users to access the FTP services. For the FTP service, an easy target for the attacker would be the buffer that stores the client's FTP command strings, as evidenced by the buffer overflow vulnerability in PCMan's FTP Server 2.0.7 [11] and the other FTP servers (e.g., TYPSoftFTP server, Ws FTP server, Typsoft FTP server, and Core FTP server) [4].

Throughout the life cycle of an APT attack, the attacker can exploit a vulnerable FTP server to achieve the following goals:

- By hacking a vulnerable FTP server in the DMZ (demilitarized zone), the APT attacker can gain his initial foothold inside the victim enterprise network.
- The APT attacker can also exploit a vulnerable internal FTP server to move laterally inside the victim enterprise network.

As our main focus is to ensure deception consistency against an APT attacker who has already been detected, the method used to identify these attacks is orthogonal to this work. For instance, given that buffer overflow attacks are a major threat vector against FTP services, we can monitor FTP users' attempts in overflowing the buffer used to store incoming FTP commands.

As APT attackers can be skilled and well-resourced hackers, we assume that they can recognize deceptive environments from inconsistent observations. We also assume that the attacker's knowledge about the FTP service is gained only from the outputs of his FTP commands so he does *not* rely on any prior knowledge about the FTP service under exploitation to identify deceptive environments. Moreover, we assume that the attacker can infer the error code of his command from its output sent by the FTP server. Although it is possible to hide such information by manipulating the output of a command, it may confuse normal FTP users; hence, it is assumed that the FTP server being protected notifies its user of the authentic error code – or lack of errors – after each FTP command is executed.

4 ENSURING DECEPTION CONSISTENCY

Our proposed system for hardening FTP services with cyber deception is illustrated in Figure 1. The FTP service, which is instrumented with threat detection capabilities, monitors the commands sent by each individual client to detect if he attempts to exploit the FTP service. If he does, the server migrates the ongoing FTP connection to a deceptive FTP service with a file system that has sensitive information redacted.

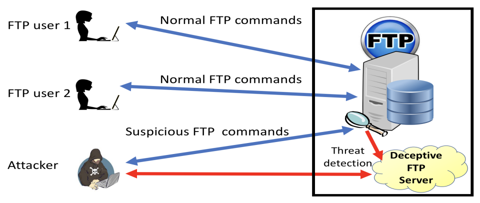


Figure 1: FTP services hardened against APT attacks

4.1 Detecting APTs against FTP services

To trap APT attackers into using deceptive FTP service, we first instrument real FTP services with threat detection capabilities. In our implementation, we have instrumented two popular FTP server-side software programs to detect attackers who attempt to send overly long command strings to overflow the input buffer:

- **Bftpd [2]:** The main function of a Bftpd server (Version 4.4) has a *while* loop and in each iteration the server parses and processes an incoming FTP command stored in variable *str*.

We instrument the code by checking whether the length of the command string *str* exceeds a maximum threshold *MAXLENGTH*, and if so, creating a deceptive virtual environment for the ongoing FTP connection.

- **ProFTPd [9]:** The main function of a ProFTPd server (Version 1.3.7rc1) contains a *while(TRUE)* loop, in which the server accepts an incoming FTP command into buffer cmd_buf. Similarly, in the loop we add code that checks if the length of cmd_buf exceeds threshold *MAXLENGTH*. An attacker sending an overly long command string will be redirected into a deceptive FTP server.

Threshold *MAXLENGTH* in both Bftpd and ProFTPd should be large enough so that it is unlikely for a normal FTP user to send a command string whose length exceeds *MAXLENGTH*. As our threat model concerns APT attackers, the simple method of detecting overly long FTP commands can still be effective in identifying persistent attackers who look for buffer overflow vulnerabilities.

Another way of identifying APT attackers is to hide decoy files inside the file system used by the real FTP service. After making it unlikely for normal users to access these files, if the real FTP server detects attempts of downloading these files, the corresponding FTP connections will be identified as threats.

4.2 Extracting APT attackers' traces

We call the process of diverting the APT attacker detected to the deceptive FTP server as a *context switch*. We now focus on the consistency of the FTP file system seen by the attacker after the context switch. Although a naive approach in which the deceptive FTP server replicates the FTP file system of the real FTP server will definitely make it consistent, the defender may not want to reveal the possibly sensitive information in the original FTP system on the host to the attacker, except those that have already been observed by the attacker. Hence, a natural solution is to simulate a different file system for the deceptive FTP server, which possibly contains decoy files with misinformation to deceive the attacker.

To ensure observation consistency after the context switch, we collect all possible traces about the APT attacker before he is trapped into the deceptive environment. Based on our threat model, the attacker gains knowledge about the files on the real FTP server only through his use of its FTP service. Hence, we instrument the log system of the real FTP server to support the following: for each FTP command received, the server records the PID and the client's IP address, the raw FTP command, the arguments of the command, the returned value after executing the command, and the output results of the command. Raw FTP commands, such as CWD (changing working directory), LIST (list remote files), and SIZE (return the size of a file), have been defined in various FTP-related RFC standards, such as RFCs 697, 959, 1639 and 2228. Our implementations only concern those raw RTP commands that are supported by the real FTP server. After a specific FTP connection is identified as suspicious, we use the attacker's IP address *AIP* to obtain the list of RAW commands, denoted by L_{AIP}, that have been executed on behalf of the attacker's FTP connection.

For ease of presentation, we introduce a few other notations here. For each command with raw command C (e.g., CWD) on list L_{AIP}, we use $(R, O) = C(I)$ to denote that given input I, executing command C returns R and generates output O. For example, assuming that the attacker listing directory `/a` which contains a file `b.txt` and a sub-directory `c`, after parsing the logs we obtain the following: C = LIST, I = "/a", R = 1 (which means the directory `/a` exists), and O =

```
[['PATH':'/a/b.txt', 'DIR':0, 'PERM':33279, 'USER'=666,
  'GROUP':666, 'MODTIME':1510002103, 'SIZE':10],
 ['PATH':'/a/c', 'DIR':1, 'PERM':16877, 'USER':666,
  'GROUP':666, 'MODTIME':1511371699, 'SIZE':4096]].
```

In this example, each element of output O is a key-value map that contains the information obtained for a subpath (a file or a sub-directory): 'DIR' indicates whether it is a directory or not (0 means no and 1 means yes), 'PERM' gives the permission of the corresponding path, 'USER' is the user ID, 'GROUP' is the group ID, 'MODTIME' records the last modification time of the sub-path, and 'SIZE' gives the size of the sub-path in bytes.

Therefore, *for deception consistency assurance, we have taken into consideration not only the structure of the FTP file system, but also the meta-data associated with the file system.* This is important, because the subtle differences between even the sizes of the same files in the real and deceptive file systems may alert the attacker.

4.3 Representing attacker's best knowledge

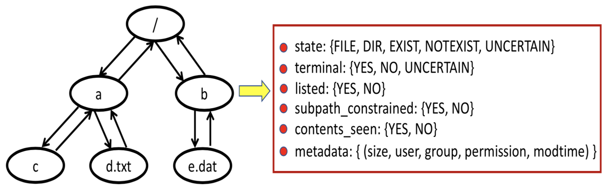

/
a
b
c
d.txt
e.dat
state: {FILE, DIR, EXIST, NOTEXIST, UNCERTAIN}
terminal: {YES, NO, UNCERTAIN}
listed: {YES, NO}
subpath_constrained: {YES, NO}
contents_seen: {YES, NO}
metadata: { (size, user, group, permission, modtime) }

Figure 2: Attacker's knowledge represented with tree T_{cstr}

We use an attributed tree T_{cstr}, as illustrated in Figure 2, to maintain the attacker's best knowledge gained from the outputs of his FTP commands as *logic constraints*. The tree is updated dynamically by iterating over each item on L_{AIP}, the list of commands obtained from the logs. The tree structure naturally reflects that of a file system, but it cannot be just the same as a file system because it needs to represent *negative knowledge* such as a nonexistent path. Hence, each node in the tree represents an atomic component in a path, either a directory or a file, and by concatenating all the nodes starting from the root, a node also represents a unique path. For example, node b in Figure 2 corresponds to a unique path `/b`. Given node v in the tree, we use $path(v)$ to denote the corresponding absolute path starting from the root. Moreover, a *subpath* of $path(v)$ is any path that starts from the root and ends with one of the parent nodes of v. For example, both `/` and `/b` are subpaths of `/b/e.dat`. With path p, we use $node(p)$ to denote the last node on p, $|p|$ the number of nodes on p, and $subpaths(p)$ a set including all subpaths of p ordered by increasing lengths. We also use $parent(v)$ to represent the parent node of v.

To represent the attacker's best knowledge as logic constraints, we augment each node v in T_{cstr} with the following attributes:

- **state**: Attribute $state(v)$ has five possible values: FILE ($path(v)$ must exist as a file), DIR ($path(v)$ must exist as a directory), EXIST ($path(v)$ must exist and can be either a file or a directory), NOTEXIST ($path(v)$ must not exist), and UNCERTAIN (the state of $path(v)$ is uncertain). The default value of attribute $state$ is UNCERTAIN.

- **terminal**: Attribute $terminal(v)$ has three values: YES ($path(v)$ must be a file or an empty directory), NO ($path(v)$ must be a nonempty directory), and UNCERTAIN (otherwise). The default value of attribute $terminal$ is UNCERTAIN.
- **listed**: Attribute $listed(v)$ is a Boolean value, which indicates if the attacker has executed a command listing $path(v)$. The default value of attribute $listed$ is NO.
- **metadata**: Attribute $metadata(v)$ maps the metadata fields of $path(v)$ to their values. The metadata attribute of a node is updated when the raw LIST command is executed by the attacker. The default value of each field is NC (no constraint).
- **subpath_constrained**: This Boolean attribute is useful for some FTP commands that return an error indicating that part of a path is a file. Following the example shown in Figure 2, if path `/a/d.txt` is a file and the attacker tries to create a sub-directory under it (e.g., by executing raw FTP command `MKD /a/d.txt`), the error code 20 is returned, suggesting that the attacker would be able to know one of the subpaths (i.e., either `/a` or `/a/d.txt`) is actually a file. The default value of attribute *subpath_constraint* is NO.
- **contents_seen**: This attribute is set to YES only after the contents of a file have been seen by the attacker, after he uploads or downloads a file; otherwise, its value is NO. The default value of attribute *contents_seen* is NO.

When a new node is created in tree T_{cstr}, its attributes are initialized to be their default values. The error codes when an FTP server parses raw FTP commands are explained in Table 2 at the end of this paper. We note that the returned values of these commands can lead to similar observations made by the attacker, such as whether a path exists, whether a subpath of a path is a file, and whether a directory is empty. We thus define four basic procedures of updating tree T_{cstr} whose pseudo code is given in Algorithm 1:

- **HANDLE_PATH_EXIST(p, s)**: this procedure is called when it is confirmed that path p must exist. If the type of path p is certain, state s is provided with the confirmed type, a file (FILE) or a directory (DIR); otherwise, type EXIST is provided as the input parameter. When this procedure is called, all subpaths of path p are also confirmed to be a directory (DIR). If it is a file, we ensure *node*(p) is a terminal node.
- **HANDLE_PATH_NOT_EXIST(p)**: this procedure is called when it is confirmed that path p does *not* exist. We process each of the ancestor nodes of *node*(p) in the tree, denoted by q, starting from the root: if it has been confirmed previously that q is a terminal node (either an empty directory or a file), or *node*(q) is *not* in tree T_{cstr} *and* its parent node has been listed (i.e., $listed(parent(node(q))$ = YES), the knowledge that path p does not exist has already been represented by the existing constraints in tree T_{cstr} and the procedure is thus aborted; otherwise, a new node is created for q with its state as UNCERTAIN. Finally, if the parent node of path p is *not* listed, we insert *node*(p) into T_{cstr} (if not in it yet), label its state as NOTEXIST, and mark it as a terminal node. If *node*(p) already exists when the procedure is called, we need to remove all children (including their descendants) from *node*(p). However, there is a subtle case where there exists such a descendant q from the tree rooted at *node*(p) that $subpath\_constraint(q)$ is YES: we need to change $subpath\_constraint(parent(p))$ to YES because we are now sure that one subpath of $parent(p)$ must be a file.
- **HANDLE_SUBPATH_IS_FILE(p)**: for some FTP commands (e.g., MKD and DELE), if one of the subpaths of the path parameter is a file, an error is returned. We process each of the ancestor nodes of *node*(p) in the tree, denoted by q, starting from the root: if the state of node q is FILE, it is unnecessary to proceed further because the knowledge has already been contained within tree T_{cstr}; otherwise we create a new node for q with its state as UNCERTAIN if it is not in tree T_{cstr} yet. Finally, we insert *node*(p) with its state as UNCERTAIN and mark its *subpath_constrained* attribute as YES.
- **HANDLE_PATH_REMOVAL(P)**: this procedure is called when we need to remove *node*(p) from its parent node. We simply change the state of the *node*(p) to NONEXIST. After doing so, if the parent node of *node*(p) has been listed *and* it becomes empty after removing *node*(p), we mark it as a terminal node; otherwise, if the parent node is not empty before the removal, it now may become empty.

Based on the four aforementioned basic procedures, we define the list of operations for each raw FTP command. We summarize these operations in Table 2, and it is noted that a constant number of basic procedures are needed for processing each raw FTP command.

4.4 Instantiating a consistent file system

Given tree T_{cstr} with the attacker's best knowledge represented as logic constraints, we instantiate a new FTP file system in two steps:

- **LIFT**: We lift the attacker's *positive* knowledge about the file system into certainty. Completely positive knowledge includes a path being a directory or a file, its metadata values, and if a path is listed, all the contents within it. Moreover, partially positive knowledge includes an existing path, either a directory or a file, and the fact that one subpath of a path must be a file. When creating the file system for the deceptive FTP server, we lift the attacker's partially positive knowledge into completely positive in a non-deterministic manner. Once the LIFT step is finished, the file system created must be free of inconsistency according to constraints in T_{cstr}.
- **EXPAND**: The expansion phase expands the file system lifted from the attacker's positive knowledge into one that achieves the defender's deception goal. For example, the defender can put a vulnerable software program in the expanded file system, which may look valuable to the attacker, so if it is installed on the attacker's own system, it creates a backdoor for the defender. It is out of the scope of this work how to expand the file system to achieve a certain deception goal.

The pseudo-code of these two steps is presented in Algorithm 2. In procedure LIFT, we use the recursive DFS (Depth-First-Search) algorithm to traverse each node in tree T_{cstr}: if it is completely positive knowledge, we replicate the contents for the deceptive FTP server. Otherwise, if a node is in state EXIST, we lift it to a file with probability P_0 and a directory with probability $1 - P_0$. For a

Algorithm 1 Four basic procedures for updating T_{cstr}

```
procedure HANDLE_PATH_EXIST($p$, $s$)                ▷ $s \in$ {FILE, DIR, EXIST}
    for $q \in subpaths(p)$ do
        Insert $node(q)$ if not existing yet, $state(node(q)) \leftarrow$ DIR
    end for
    if $node(p)$ does not exist in $T_{cstr}$ then
        Insert $node(p)$ to $T_{cstr}$, $state(node(p)) \leftarrow$ EXIST
    end if
    if $s$ is FILE then
        $state(node(p)) \leftarrow s$, $terminal(node(p)) \leftarrow$ YES
    else if ($s$ is DIR) or ($state(node(p)) \notin$ {FILE, DIR, EXIST}) then
        $state(node(p)) \leftarrow s$
    end if
end procedure
procedure HANDLE_PATH_NOT_EXIST($p$)                  ▷ $p$ is a path
    for $q \in subpaths(p)$ do
        if $node(q)$ exists in $T_{cstr}$ then
            Quit procedure if $terminal(node(q))$ is YES
        else if $listed(parent(node(q)))$ is YES then
            Quit procedure due to certainty of $parent(node(q))$
        else
            Insert $node(q)$ with $state(node(q))$ as UNCERTAIN
        end if
        $lastq \leftarrow node(q)$
    end for
    if $listed(lastq)$ is NO then
        Insert $node(p)$ if it does not exist in $T_{cstr}$
        $state(node(p)) \leftarrow$ NOTEXIST, $terminal(node(p)) \leftarrow$ YES
        if $subpath\_constrained(w)$ is YES where $w$ is a descendant
    of $node(p)$ then
            $subpath\_constrained(parent(node(p)) \leftarrow$ YES
        end if
        Remove all children of $node(p)$, including their descendants
    end if
end procedure
procedure HANDLE_SUBPATH_IS_FILE($p$)                 ▷ $p$ is a path
    for $q \in subpaths(p)$ do
        if $node(q)$ exists in $T_{cstr}$ then
            Quit procedure if $state(node(q))$ is FILE
        else
            Insert $node(q)$ with its state as UNCERTAIN
        end if
    end for
    Quit procedure if $node(p)$ exists and $state(node(p))$ is FILE
    Insert $node(p)$ to $T_{cstr}$ if $node(p)$ does not exist
    $subpath\_constrained(node(p)) \leftarrow$ YES
end procedure
procedure HANDLE_PATH_REMOVAL($p$)                    ▷ $p$ is a path
    $state(node(p)) \leftarrow$ NOTEXIST
    if $terminal(parent(node(p)))$ is NO then
        $terminal(parent(node(p))) \leftarrow$ UNCERTAIN
    end if
    if $listed(parent(node(p)))$ is YES then
        if $parent(node(p))$ doesn't have children nodes then
            $terminal(parent(node(p))) \leftarrow$ YES
        end if
    end if
end procedure
```

directory that is not a terminal node in T_{cstr} but does not have any children nodes in state DIR or FILE, we add a new single file (with probability P_1) or sub-directory (with probability 1 - P_1) in it so that the constraint can be satisfied. Another subtle case occurs when a node is in state UNCERTAIN and its *subpath_constrained* attribute is YES, which means that one of the subpaths (including itself) is a file. With probability P_2, the node itself is lifted to a file, and with probability $1 - P_2$, the path corresponding to the node does not exist and one subpath of the parent node is a file. In the latter case, we remove the node and change the parent node's *subpath_constrained* attribute to YES. However, it is possible that a parent node may have multiple children nodes whose *subpath_constrained* attributes are set to YES. In that case, we prioritize the lifting operations based on the processing order. If processing one child node lifts it to a file so its parent node must be a directory, the other children nodes must also be files. On the other hand, if processing one child node lifts it to a non-existing node so the *subpath_constrained* attribute of its parent node becomes YES, it means that the parent node itself or one of its subpaths must be a file and therefore all its other children nodes must not exist.

The three probability parameters, P_0, P_1 and P_2, during the LIFT step offer non-determinism in instantiating a new FTP file system for the deceptive FTP server. Non-determinism is important because otherwise, the attacker may realize the deceptive environment. For example, if he notices that an error of a non-existing path is always correlated to the existence of its parent path, this surely looks suspicious to him, suggesting that a deterministic algorithm that always lifts a *non-existing path* to an *existing parent path* is not ideal. Currently all three probability parameters are set to 1/2 by default.

When instantiating the FTP file system for the deceptive FTP server during the LIFT step, we need to ensure that the metadata fields of the files or directories created should be consistent with their corresponding constraints stored in tree T_{cstr}. If the *contents_seen* attribute of a path is YES, we need to ensure that if the corresponding file on the deceptive FTP server is requested, a duplicate of the original one should be returned. If the size of a file has been seen by the attacker, the file created for the deceptive server should have the same size. In Section 4.5, we will discuss a number of techniques that can speed up the process of instantiating the FTP file system.

The EXPAND procedure also uses the recursive DFS algorithm to traverse nodes in tree T_{cstr}. Obviously only directories need to be expanded, and a directory that has been listed or is confirmed to be a terminal node cannot be expanded any more. While expanding a directory, we check the corresponding constraints in T_{cstr} to ensure that the simulated files or sub-directories should *not* conflict with the attacker's *negative* knowledge about the file system, corresponding to those nodes in the NOTEXIST state.

4.5 Performance optimization

It may take a long time to create the entire FTP file system from scratch for the deceptive FTP server. If the attacker's FTP connection is restored thereafter, the attacker may identify the deceptive environment due to the long latency after the attack command is sent to the server. We apply the following techniques to shorten the context switch period.

(1) Deferred-expansion technique: We restore the attacker's FTP connection immediately after the LIFT step, and the context switch process will keep expanding the file system until the entire file system is created. This may lead to some inconsistency issues, if part of the file system explored by the attacker immediately after the context switch has not been expanded yet. Therefore, until the file system is fully expanded, the context switch process will keep updating a separate constraint tree T'_{cstr} with new constraints generated from the attacker's FTP commands executed on the deceptive FTP server, and constraints in both T_{cstr} and T'_{cstr} are used to avoid conflicts while expanding the FTP file system.

(2) Pre-generation technique: Instead of creating every part of the FTP file system from scratch on demand, some directories can be pre-generated for the deceptive FTP server. Later during the EXPAND operation, if a directory needs to be created, a pre-generated directory is randomly chosen, based on its level in the file system, and then hooked into the expanded FTP file system.

(3) Request-forwarding technique: If the *contents_seen* attribute of a path is YES and if the corresponding file is large, copying the file to the deceptive FTP server may be time consuming during the context switch. To reduce the overhead, we can let the deceptive FTP server forward the file transfer request to the original FTP server, which works as a proxy in responding to the file transfer request.

(4) File bloating technique: When creating a file of a specific size, a naive method that truncates a larger file to the correct size may cut off the file, leading to inconsistent file contents. A better approach is to extend an existing file that is slightly smaller than the target size with unused padded bytes. For example, the *truncate* utility in Linux can extend a file to a specified size efficiently because it creates a sparse file without allocating real physical blocks for the padded bytes [13].

5 IMPLEMENTATION

The architecture illustrated in Figure 1 can be implemented in a variety of ways: the real and deceptive FTP servers can be deployed on different physical hosts, different VMs on the same physical host, different VMs on different physical hosts, different containers on the same physical host, different containers on different physical hosts, and so on. Each of these implementations has its own advantages and challenges. This section presents the specific challenges we have faced when deploying the real FTP server on the host and the deceptive FTP server in a VM on the same host, as well as our solutions to address these challenges. We choose this implementation because it does not need an extra physical machine to host the deceptive FTP server and nowadays VMs are hard to escape even if the attacker manages to break out of the jailed FTP service.

5.1 Migrate attacker's FTP connection into a VM on the same host

When we implement the context switch by redirecting an ongoing attack FTP connection to a deceptive FTP server within a VM on the same host, we need to address two types of *transparency* challenges:

- **Service transparency:** The FTP service provided by the deceptive FTP server should continue without noticeable changes of the underlying network status to the attacker. After the context switch, therefore, the attacker's FTP client

Algorithm 2 Instantiate a file system based on T_{cstr} and T'_{cstr}

```
Require: $T_{cstr}$, three configurable probabilities $P_0$, $P_1$, and $P_2$
 1: Step 1: LIFT($T_{cstr}$, $root(T_{cstr})$)
 2: Step 2: EXPAND($T_{cstr}$, $root(T_{cstr})$)
 3:
 4: procedure LIFT($T$, $v$)                    ▷ $T$ is a tree and $v$ is a node
 5:     for each child node $w$ of $v$ in $T$ do
 6:         Call procedure LIFT($T$, $w$)
 7:     end for
 8:     if $state(v)$ is NOTEXIST then
 9:         Continue                                              ▷ Do nothing
10:     else if $state(v)$ is EXIST then    ▷ parent node of $v$ must be DIR
11:         $r \leftarrow$ a random number in [0, 1]
12:         if $r \le P_0$ then
13:             $state(v) \leftarrow$ FILE, $terminal(v) \leftarrow$ YES
14:         else
15:             $state(v) \leftarrow$ DIR
16:         end if
17:     else if $state(v)$ is FILE or DIR then
18:         $state(parent(v)) \leftarrow$ DIR, $terminal(parent(v)) \leftarrow$ NO
19:         if $state(v)$ is DIR and ($listed(v)$ is NO) and ($terminal(v)$ is
            NO) and ($\nexists w \in children(v) : state(w) =$ DIR or FILE) then
20:             $r \leftarrow$ a random number in [0, 1]
21:             if $r \le P_1$ then    ▷ Add something to satisfy the constraint
22:                 Add a new file in $path(v)$
23:             else
24:                 Add a new sub-directory in $path(v)$
25:             end if
26:         end if
27:     else if $state(v)$ is UNCERTAIN then
28:         if ($subpath\_constrained(v)$ is YES) then
29:             if ($subpath\_constrained(parent(v))$ is NO) then
30:                 if $state(parent(v))$ is UNCERTAIN then
31:                     $r \leftarrow$ a random number in [0, 1]
32:                     if $r \le P_2$ then                     ▷ it is a file
33:                         $state(v) \leftarrow$ FILE, $terminal(v) \leftarrow$ YES
34:                         $state(parent(v)) \leftarrow$ DIR
35:                         $terminal(parent(v)) \leftarrow$ NO
36:                     else                                 ▷ it doesn't exist
37:                         $subpath\_constrained(parent(v)) \leftarrow$ YES
38:                         Remove $v$ from children of $parent(v)$
39:                     end if
40:                 else                              ▷ This needs to be a file
41:                     $state(v) \leftarrow$ FILE, $terminal(v) \leftarrow$ YES
42:                 end if
43:             end if
44:         end if
45:     end if
46: end procedure
47: procedure EXPAND($T$, $v$)      ▷ $T$ is a tree and $v$ must be a directory
48:     for each node $w$ where $w \in children(v)$ and $state(w) =$ DIR do
49:         Call procedure EXPAND($T$, $w$)
50:     end for
51:     if ($listed(v)$ is NO) and ($terminal(v)$ is not YES) then
52:         Simulate files or sub-directories inside $path(v)$ without con-
            flicting with constraints in $T_{cstr}$ or $T'_{cstr}$
53:     end if
54: end procedure
```

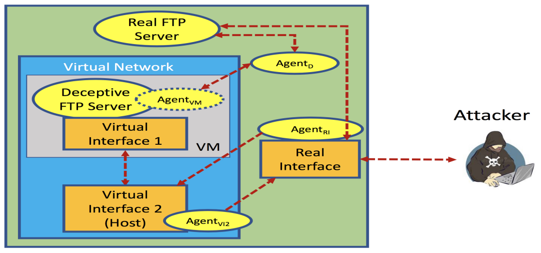


Figure 3: Deceptive FTP service within a VM on same host

should be able to continue communicating with *the same destination IP address* of the real FTP server, although actually talking to the deceptive FTP server in the VM.

- **Migration transparency:** The process of migrating the attacker's FTP connection to the deceptive FTP server should be done with low latency so that the attacker cannot tell whether process migration has occurred.

Our implementation is illustrated in Figure 3. Without loss of generality, we walk through an example in which the real FTP server runs on a host with a NIC (Network Interface Card) with public IP address *RIP* and MAC address *RMAC*.

During the **initialization phase**, which occurs before the attacker's FTP connection is detected, the host creates a placeholder VM for the deceptive FTP server. The VM is connected to a virtual network created on the host machine through a virtual interface with IP address VIP_1 and MAC address $VMAC_1$. The host itself is connected to the virtual network through a virtual interface with IP address VIP_2 and MAC address $VMAC_2$. During the initialization phase, the deceptive FTP server is started to accept new FTP connections (the reason will be explained later in Step VB3). Inside the VM, there is also an *ephemeral agent*, $Agent^{VM}$, which expects to contain a new attacker's FTP connection.

After the attacker's FTP connection is detected, it is migrated from the host machine into the VM and then served by the deceptive FTP server. This process, which is called the **context switch phase**, consists of the following steps:

- **CW1:** The real FTP server obtains the PID (Process Identifier) of the attacker's FTP connection and also his IP address (AIP), and then notifies the defense agent $Agent^D$, which runs on the host machine, of both the PID and the AIP.
- **CW2:** On receiving these information, $Agent^D$ dumps all the states associated with process PID into an image file, using the CRIU utility [3].
- **CW3:** After dumping the attacker's FTP connection, $Agent^D$ immediately adds a new iptables rule on the host machine dropping packets sent from the attacker's IP address AIP. Otherwise, the real FTP server may reply with a rejection packet that looks suspicious to the attacker.
- **CW4:** The image file is then compressed and transmitted to $Agent^{VM}$ inside the VM through its IP address VIP_1. Along with the attacker's process image file, $Agent^D$ also sends a list of logic constraints to $Agent^{VM}$.
- **CW5:** After transmitting all the files to the VM, $Agent^D$ adds another iptables rule that drops all the packets from the VM with source IP addresses as VIP_1.

In the next stage (the **VM bootstrapping phase**) $Agent^{VM}$ restores the attacker's FTP connection and also carefully changes the state of the VM so that even if the attacker breaks out of the deceptive FTP service, he still cannot recognize the contained deceptive environment. More specifically $Agent^{VM}$ performs the following:

- **VB1:** $Agent^{VM}$ changes the IP address of the VM (i.e., virtual interface 1 in Figure 1(2)) from VIP_1 to that of the host *RIP*. This is necessary because the attacker's FTP connection still assumes that the FTP server's IP address is *RIP*.
- **VB2:** We next need to ensure that the VM can communicate with the attacker after its IP address is changed to *RIP*. A standard OS kernel typically delivers a packet as follows. It first checks whether the destination IP address is in the same network. If so, the ARP protocol is used to find the MAC address associated with the destination IP address. Otherwise, it uses the routing table to find the route for the packet based on the destination IP address. If no such route is found, the default route is used to forward the packet. As we do not plan to change the OS kernel in this work, we consider the following two cases:
  - **Case VB2(a):** The attacker's source IP address does not belong to the same subnet as the real FTP server. This can occur when the attacker tries to exploit the FTP service from outside the enterprise network, hoping to find his initial foothold inside the network. In this case, we add a default route inside the VM to forward every packet to a gateway. The IP address of the gateway, denoted as *GIP*, is set to be the same as that of the host machine's gateway, if it exists, or an arbitrary one that shares the same network mask with *RIP* but is different from *RIP*. The purpose of this step is to prevent the attacker from finding out the deceptive environment based on an unrealistic gateway IP address *GIP* used inside the VM. Afterwards, $Agent^{VM}$ adds a static ARP rule that binds IP address *GIP* to $VMAC_2$, the virtual MAC address through which the host machine is attached to the same virtual network as the VM.
  - **Case VB2(b):** The attacker's source IP address belongs to the same subnet as the real FTP server. This happens when the attacker attempts to compromise an internet FTP server for its lateral movement inside the enterprise network. In this case, setting a default route does not work because the OS kernel at the data link layer uses the ARP protocol to find the destination MAC address. Hence, we create a static ARP rule that binds the attacker's IP address AIP directly to $VMAC_2$ inside the VM.
- **VB3:** Using CRIU [3], $Agent^{VM}$ restores the attacker's FTP connection inside the VM from the image file dumped from the real FTP service by $Agent^D$. Recall that during the initialization phase, an FTP server is started to accept new FTP connections inside the VM. This running FTP server is crucial for two reasons. First, the attacker may log out and then re-log into the server. If he cannot log into the server again, he may recognize the deceptive environment. Second, for some FTP client software (e.g., gftp [6]), when the user uploads or downloads a file, instead of using the original FTP connection, they create another one to deal with file transfer.

In such cases, if there is no FTP server running, the attacker will not be able to upload or download files, which makes him suspicious of the deceptive environment.

- **VB4:** $Agent^{VM}$ kills itself so the attacker cannot find the process of $Agent^{VM}$ even if he can break out of the deceptive FTP server. This is the reason why $Agent^{VM}$ is ephemeral. This step is done instantly after $Agent^{VM}$ finishes its tasks to minimize the likelihood of a race condition that the attacker can exploit to find the existence of $Agent^{VM}$.

While $Agent^{VM}$ performs bootstrapping inside the VM, $Agent^{D}$ creates two other agents on the host, $Agent^{RI}$ and $Agent^{VI2}$, to perform packet address translation:

- $Agent^{RI}$: Working at the MAC layer, $Agent^{RI}$ monitors each packet on the real interface *RMAC* sent from the attacker's IP address *AIP*, changes its destination MAC address to $VMAC_1$ (i.e., the virtual interface of the VM), and then resends the packet through virtual interface $VMAC_2$.
- $Agent^{VI2}$: Also working at the MAC layer, $Agent^{VI2}$ monitors each packet on virtual interface $VMAC_2$ (i.e., the one through which the host attaches to the virtual network), and if a packet is destined to the attacker's IP address, $Agent^{VI2}$ changes its source MAC address to the real interface *RMAC* and sends it to the attacker through the real interface *RMAC*.

After the attacker's FTP connection is migrated into the VM, the transmission time of a packet between the deceptive FTP server and the real NIC of the host differs from that between the real FTP server and the real NIC. As the VM is located within the same physical host machine as the real FTP server, the migration latency is small compared to that where attackers' connections are migrated onto different physical machines.

The four agents, $Agent^{D}$, $Agent^{RI}$, $Agent^{VI2}$ and $Agent^{VM}$ are implemented in Python. Root privilege is needed by $Agent^{D}$ to dump the attacker's FTP connection from the real FTP server on the host machine, and by $Agent^{VM}$ to restore the connection, add the default route for packet forwarding, and create a static ARP binding rule inside the VM. $Agent^{RI}$ and $Agent^{VI2}$ use Scapy [10] to monitor, rebuild, and send packets on network interfaces directly; because they need to interact with the network stack in the OS kernel, they are also executed with root privilege on the host.

5.2 Other implementation issues

CRIU. We have used CRIU version 2.6 for live migration of the attacker's FTP connection. Following the tutorial of CRIU, the process of using CRIU to dump and restore FTP connections in Bftpd is straightforward. But when doing the same thing on ProFTPd, we encounter an error: *"Error (criu/pie/parasite.c:339): can't dump unpriviliged task whose /proc doesn't belong to it."* Examining the source code of ProFTPd, we find that ProFTPd jails each FTP connection by changing the root directory of its corresponding process, but CRIU cannot access the files in the system directory `/proc`, which leads to the aforementioned error. To circumvent the problem, we copy both the `fd` directory and the `cgroup` file in the system directory `/proc/$PID` to `/proc/$PID` in the jailed file system, create a soft link with name `self` at `/proc` in the jailed file system, and then change the permission of `/proc/` in the jailed file system to be 777 (i.e., readable, writable, and executable by everyone). After the FTP connection is restored inside the VM by $Agent^{VM}$, the entire directory `/proc` is deleted from the FTP file system.

Another issue when migrating the FTP connection for ProFTPd is that three files (PFTEST.pid, PFTEST.score, PFTEST.score.lck) are not dumped by CRIU, but their existences are checked by CRIU when restoring the FTP connection. The problem can be solved by copying these three files from the host machine into the VM.

Virtualization. We have chosen KVM/QEMU to create VMs and virtual networks as they are full integrated into Linux. The virtual network created as seen in Figure 3 is isolated from any physical network to which the host machine belongs. The VM cannot use NAT (Network Address Translation) to communicate with the outside network as in the default user-mode networking configuration, because its network interface shares the same IP address as that of the host machine. Instead, our proposed system relies on address manipulation at the data link layer by the two agents, $Agent_{RI}$ and $Agent_{VI2}$, to enable communications between the VM and the attacker's machine.

As the real FTP server runs on a host machine while the deceptive one in a VM, one inherent source of inconsistency is virtualization latency: the response time per command experienced by the attacker when interacting with the deceptive FTP server inside the VM is longer than that with the real FTP server. This issue does not exist in some other implementations (e.g., both the real and the deceptive FTP servers run inside VMs). To reduce latency inconsistency, we make up the difference within the real FTP server by delaying processing each command for a period that is close to the virtualization latency. The delay can be configured based on the hardware support for virtualization.

6 EXPERIMENTS

In our experiments, we evaluate the effectiveness of our methods in deceiving APT attackers. We also measure the virtualization latency due to the necessity of containing the attackers within VMs, and the migration latency during the context switch.

6.1 Deception experiments

We evaluate the effectiveness of our method in deceiving simulated APT attackers. We use two laptops, both of which have Ubuntu 16.04.3 LTS installed. One laptop is used to run the native FTP client provided by Ubuntu Linux, and the other runs a ProFTPd FTP server. The two laptops are located in the same office room, using a campus wireless network for communications. The FTP server runs on a Dell Inspiron 15 5800 Series Laptop, which has the dual-core Intel i7-5500U CPU and 64GB RAM, and the same OS (Ubuntu 16.04.3 LTS) is used for both the host machine and the VM.

In our experiments, we consider two types of FTP file systems:

- **Project file system**: The FTP file system is populated with real files and directories extracted from the ProFTPd project.
- **Random file system**: The FTP file system is created with random directories, each populated with random files.

For the deceptive FTP server, we use the instantiation algorithm described in Section 4.4 to create its file system: if the random file system is used for the real FTP server, the file systems used by the real and deceptive FTP servers share similar ratios of the numbers of files to that of sub-directories inside the directories at the same level; otherwise, the simulated file system used by the deceptive

Table 1: Evaluation results from two sets of experiments. X/Y means a student's answer is X and the ground truth is Y. 'Yes' means that deception is enabled and 'No' no deception.

Group ID	#students	Level	Exp. 1	Exp. 2
1	1	undergraduate	No/No	Yes/Yes
2	1	undergraduate	No/No	No/Yes
3	1	graduate	No/No	Yes/No
4	1	graduate	No/No	No/Yes
5	3	graduate	No/No	Yes/Yes
6	3	graduate	No/Yes	No/Yes
7	3	undergraduate	No/Yes	No/Yes
8	1	undergraduate	Yes/No	Yes/No
9	1	undergraduate	No/No	No/No
10	1	undergraduate	No/No	No/Yes
11	1	undergraduate	Yes/No	No/No
12	3	undergraduate	Yes/No	Yes/Yes
13	3	undergraduate	Yes/Yes	No/No
14	3	graduate	Yes/Yes	No/No
15	3	undergraduate	No/No	No/Yes
16	3	graduate	Yes/Yes	Yes/Yes

FTP server has a similar distribution of the number of files per type in a directory (e.g., the numbers of C source and header files).

The APT attacks are simulated by 32 students (18 undergraduate and 14 graduate) who completed a computer security course. These students form 16 groups, as shown in Table 1. As we do not limit each group's experiment time, the time each group spent ranged from 10 minutes to an hour (including both experiments). We provide a list of around 20 sample FTP commands in case that the students are not familiar with FTP commands. These commands include *ls*, *cd*, *delete*, *mkdir*, *rename*, *put*, and *get*, and they can lead to both positive and negative knowledge about the FTP file system. The participants are encouraged to use the other commands, if they help them identify the deceptive environment. Among these commands, one is overly long and will be treated as a threat by the FTP server. Once such a command is detected, *with probability 1/2* the FTP server switches to the deceptive mode where the live FTP connection is migrated into the VM. For each group of participants, we have two experiments, one using the ProFTPd project file system, and the other a random file system as described earlier.

In each experiment, the participants observe the outputs of their FTP commands, including an overly long one that is deemed as a buffer overflow attack by the server. At the end of each experiment, they need to decide if the server has switched to a deceptive mode after their attack. Participants are allowed to write down what they have seen on the computer screen, and a few teams even took pictures of the computer screen for close examination later. For groups with multiple members, they are free to discuss but the majority of their votes is used as their final decision.

The results are summarized in Table 1. It is noted that even if the participants guess randomly without seeing their FTP command outputs, their accuracy is expected to be 50%. In our experiments, the accuracy for the first experiment with the project file system is 68.8% and the second one with a random file system is 50%.

From the experiments, we also make the following observations:

- *Consistency is crucial for hiding the deceptive environments.* Participants rely heavily on signals that have changed after the attack to decide whether deception has occurred. For instance, participants in Group 5 noticed that in Experiment 1, before the attack, the original file system has one file whose permission characters include 't' (temporary file), but after the attack, this permission character is not found for the same file. After seeing this, they immediately concluded that deception must have occurred. We fixed the bug after their experiments, and none of the groups after their experiments could use it again to identify the deceptive environment.
- *When no clear inconsistency observed, participants tend to think no deception occurs.* The fraction of no deception answers is 62.5%, which is significantly higher than the fraction of experiments with deception disabled (53.1%). This psychological effect may benefit the defenders because with improved cyber deception methodologies, the attackers tend to think that no deception should occur.

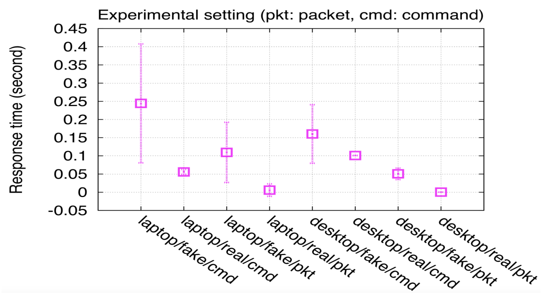
6.2 Virtualization latency

In the host-to-VM scenario, a delay closed to the virtualization latency needs to be added by the real FTP server in its FTP command processing so the attacker experiences a similar round-trip delay after the context switch. We perform experiments to measure the response time per FTP command experienced by a simulated attacker, when he interacts with the real FTP server on the host or the deceptive FTP server inside the VM. As the response time may change with the FTP command, we use a random set of FTP commands of different types in the experiments. Moreover, a FTP command can generate multiple packets, so we also measure the response time at the packet level. One set of experiments uses the same two laptops as Section 6.1, and the other two workstations located within in the same office.

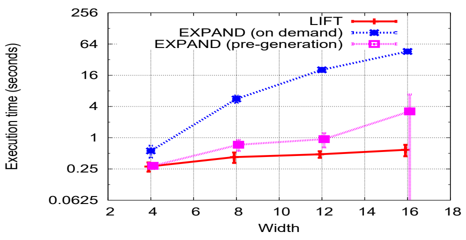
Figure 4(1) depicts the response times in different settings. Between the two laptops, the average FTP command response time and the packet-level response time differ by 200 and 100 milliseconds due to virtualization, respectively. When the experiments are performed between two desktops, both the average FTP command response time and the packet-level response time differ by 50 milliseconds. The smaller response times between the desktops result from their higher processing power. As in an enterprise network FTP services are usually deployed on dedicated desktop servers, the extra delay of 50 milliseconds per command should be acceptable to normal FTP users.

6.3 Migration latency

To measure migration latencies, we perform experiments using an FTP file system with a regular five-level structure (including the root). We assume that the root is at level 0. We define a width parameter, w, to be the fixed number of entries in any directory. Assuming that there are more directories at lower levels and more files at higher levels, we let the number of directories at level k in a directory be $w/2^{k-1}$. For instance, if $w = 8$, the root directory contains 8 sub-directories and no files, and a directory at level 1 has four sub-directories and four files. In each experiment, we execute 20 commands randomly sampled from [`ls`, `cd`, `mkdir`, `rm`, `delete`, `rename`, `get`] (with replacement) and execute them from



(1) Response time



(2) Migration latency

Figure 4: Latency results in the experiments. The error bars show the standard deviations.

the FTP client side. For each `path` argument of a command (command `rename` requires two path arguments), we recursively generate it starting from the root: with probability 1/2 the current path is appended with a number chosen from $[1, w]$ and then returned, and with another probability 1/2 a number randomly chosen from $[1, w + 2]$ is appended to the current path and the process continues at the next level. Using this scheme, it is possible that a non-existing path is created, either due to choosing a file as part of a subpath or choosing a non-existing subpath in the current directory.

Figure 4(2) presents the execution times of the LIFT and EXPAND procedures when varying the width parameter w; for the EXPAND procedure, it also gives the results in two cases depending on whether the pre-generation scheme is used or not (see Section 4.4). Clearly, the execution times of both LIFT and EXPAND increase with parameter w. Larger w implies a larger file system with more files and directories, making it longer to create a new file system for the deceptive FTP server inside the VM. Moreover, the LIFT procedure is short (less than 600 milliseconds in our experiments), regardless of w. Recall that the attacker's FTP connection is restored immediately after the LIFT procedure is completed. Hence, the migration latency which may be visible to the attacker is very short. It is also observed that pre-generating part of the file system significantly shortens the time to instantiate the entire file system inside the VM, which is helpful when there is a race between the attacker's continuing exploration of the deceptive FTP file system and the defender's creation of (mis-)information to deceive the attacker.

7 CONCLUSIONS

This work studies how to use deception to protect FTP services from APT attacks. We develop methods to ensure observation consistency, which makes it hard for APT attackers to detect the deceptive environment. With participants with computer security training for a full semester, our experiments have demonstrated the crucial importance of consistency in identifying deceptive environments.

Although making it more difficult for APT attackers to recognize deception, our proposed method is not a panacea for thwarting all types of APT attacks. Like any other defensive technology, it incurs extra overhead and may be subverted by more advanced attacks. In the future, we will continue to improve the effectiveness of deception-based methods against APTs in different attack scenarios and also investigate the economics of deception-based methods: will the defense benefits of a cyber deception technique be worthy of its deployment cost in practice?

ACKNOWLEDGMENT

We acknowledge NSF Award CNS-1618631 for supporting this work.

REFERENCES

[1] https://www.equifaxsecurity2017.com/.
[2] Bftpd. http://bftpd.sourceforge.net/.
[3] CRIU. https://criu.org/.
[4] CVE Details. https://www.cvedetails.com/.
[5] File Transfer Protocol. https://en.wikipedia.org/wiki/File_Transfer_Protocol.
[6] gFTP. https://www.gftp.org/.
[7] Kippo - SSH Honeypot. https://github.com/desaster/kippo.
[8] Papers from the Honeynet project. https://www.honeynet.org/papers.
[9] ProFTPD. http://www.proftpd.org/.
[10] Scapy. http://www.secdev.org/projects/scapy/.
[11] CVE-2013-4730. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4730, 2013.
[12] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of ACM CCS'14*, 2014.
[13] Archlinux. Sparse file. https://wiki.archlinux.org/index.php/sparse_file.
[14] K. Borders, L. Falk, and A. Prakash. Openfire: Using deception to reduce network attacks. In *Proceedings of SecureComm'07*. IEEE, 2007.
[15] B. Cheswick. An evening with berferd in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference*, 1992.
[16] F. Cohen. The use of deception techniques: Honeypots and decoys. 3, 2006.
[17] P. Ford-Hutchinson. Securing ftp with tls (rfc 4217). 2005.
[18] X. Han, N. Kheir, and D. Balzarotti. Evaluation of deception-based web attacks detection. In *Proceedings of ACM Workshop on Moving Target Defense*, 2017.
[19] M. Horowitz and S. Lunt. Ftp security extensions (RFC 2228), 1997.
[20] S. Jajodia, V. Subrahmanian, V. Swarup, and C. Wang. *Cyber Deception: Building the Scientific Foundation.* Springer, 2016.
[21] J. Jones. Cyber deception via system manipulation. In *Proceedings of the 12th International Conference on Cyber Warfare and Security*, 2017.
[22] M. Korolov. Deception technology grows and evolves. https://www.csoonline.com/article/3113055/security/deception-technology-grows-and-evolves.html.
[23] K. McCoy. Target to pay $18.5M for 2013 data breach that affected 41 million consumers. https://www.usatoday.com/story/money/2017/05/23/target-pay-185m-2013-data-breach-affected-consumers/102063932/.
[24] V. Neagoe and M. Bishop. Inconsistency in deception for defense. In *Proceedings of the 2006 Workshop on New Security Paradigms*. ACM, 2006.
[25] T. H. Project. Sebek: A kernel based data capture tool, 2003.
[26] N. Provos. Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2, page 4, 2003.
[27] N. Rowe, H. Goh, S. Lim, and B. Duong. Experiments with a testbed for automated defensive deception planning for cyber-attacks. In *Proceedings of the 2nd International Conference on I-Warfare and Security (ICIW'07)*, 2007.
[28] N. C. Rowe. Deception in defense of computer systems from cyber attack. *Cyber Warfare and Cyber Terrorism*, page 97, 2007.
[29] J. Sun and K. Sun. Desir: Decoy-enhanced seamless ip randomization. In *Proceedings of INFOCOM'16*. IEEE, 2016.
[30] J. Sun, K. Sun, and Q. Li. Cybermoat: Camouflaging critical server infrastructures with large scale decoy farms. In *Proceedings of IEEE CNS'17*. IEEE, 2017.
[31] J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *Journal of Information Warfare*, 5(3), 2006.
[32] J. J. Yuill. Defensive computer-security deception operations: Processes, principles and techniques. In *Ph.D. Dissertation, North Carolina State University*, 2006.

Table 2: Operations performed on tree T_{cstr} to maintain attacker's knowledge as logic constraints

Command	Returned value R	Meaning	Operations performed on tree T_{cstr} to maintain logic constraints
LIST($path$)	-1	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	0	$path$ is a file	HANDLE_PATH_EXIST($path$, FILE).
	1	$path$ is a directory	HANDLE_PATH_EXIST($path$, DIR); $listed(node(path))$ = YES; for each entry e in the output, insert a new node v, if not in T_{cstr} yet, and update attributes: $state(v)$ = DIR if e.DIR = 1, or $state(v)$ = FILE and $terminal(v)$ = YES if e.DIR=0; $metadata(v)[permission]$ = e.PERM; $metadata(v)[user]$ = e.USER; $metadata(v)[group]$ = e.GROUP; $metadata(v)[size]$ = e.SIZE; $metadata(v)[modtime]$ = e.MODTIME.
CWD($path$)	0	$path$ exists	HANDLE_PATH_EXIST($path$, DIR).
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	20	$path$ is a file	HANDLE_PATH_EXIST($path$, FILE).
USER($path$) GROUP($path$)	0	$path$ exists	HANDLE_PATH_EXIST($path$, DIR).
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	20	$path$ is a file	HANDLE_PATH_EXIST($path$, FILE).
MKD($path$)	1	parent is a directory and $path$ doesn't exist yet	HANDLE_PATH_EXIST($path$, DIR); $terminal(node(path))$ = YES; $terminal(parent(node(path)))$ = NO; $listed(node(path))$ = YES; update the metadata fields of $node(path)$.
	2	parent doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	17	$path$ already exists	HANDLE_PATH_EXIST($path$, EXIST).
	20	subpath is a file	HANDLE_SUBPATH_IS_FILE($parent(path)$).
RMD($path$)	1	successfully remove a directory	HANDLE_PATH_EXIST($path$, EXIST); HANDLE_PATH_REMOVAL($path$); HANDLE_PATH_NOT_EXIST($path$).
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	20	subpath is a file	HANDLE_SUBPATH_IS_FILE($path$).
	39	directory not empty	HANDLE_PATH_EXIST($path$, DIR); $terminal(node(path))$ = NO.
STOR($path$) APPE($path$)	0	successfully upload or append a file	HANDLE_PATH_EXIST($path$, FILE); $terminal(node(path))$ = YES; $contents\_seen(node(path))$ = YES.
	21	$path$ is a directory	HANDLE_PATH_EXIST($path$, DIR).
	13	permission denied	HANDLE_PATH_EXIST($path$, FILE); $permission(node(path))$ = NO_WRITE
RETR($path$)	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	21	$path$ is a directory	HANDLE_PATH_EXIST($path$, DIR).
	0	successfully download a file	HANDLE_PATH_EXIST($path$, FILE); $terminal(node(path))$ = YES; $contents\_seen(node(path))$ = YES.
DELE($path$)	0	file deleted successfully	HANDLE_PATH_EXIST($path$, FILE); HANDLE_PATH_REMOVAL($path$);
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	20	subpath is a file	HANDLE_SUBPATH_IS_FILE($parent(path)$).
	21	$path$ is a directory	HANDLE_PATH_EXIST($path$, DIR).
RNFR($path$)	1	$path$ exists	HANDLE_PATH_EXIST($path$, EXIST).
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	20	subpath is a file	HANDLE_SUBPATH_IS_FILE($parent(path)$).
RNTO($oldp$, $newp$)	0	successfully change $oldp$ to $newp$	HANDLE_PATH_EXIST($oldp$, EXIST); HANDLE_PATH_EXIST($newp$, EXIST); copy attributes in $node(oldp)$ into $node(newp)$; HANDLE_PATH_REMOVAL($oldp$).
	39	both are directories but $newp$ is not empty	HANDLE_PATH_EXIST($oldp$, DIR); HANDLE_PATH_EXIST($newp$, DIR); $terminal(node(newp))$ = NO.
	2	parent path doesn't exist	HANDLE_PATH_NOT_EXIST($parent(newp)$)
	20	rename directory to file	HANDLE_PATH_EXIST($oldp$, DIR); HANDLE_PATH_EXIST($newp$, FILE).
	21	rename file to directory	HANDLE_PATH_EXIST($oldp$, FILE); HANDLE_PATH_EXIST($newp$, DIR).
MDTM($path$, new_time)	0	successfully change time	HANDLE_PATH_EXIST($path$, FILE); $metadata(node(path))[modtime] = new\_time$.
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	22	$path$ is directory	HANDLE_PATH_EXIST($path$, DIR).
SIZE($path$, new_size)	0	successfully change size	HANDLE_PATH_EXIST($path$, FILE); $metadata(node(path))[size] = new\_size$.
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	22	$path$ is directory	HANDLE_PATH_EXIST($path$, DIR).
CHMOD($path$, new_perm)	0	successfully change permission	HANDLE_PATH_EXIST($path$, FILE); $metadata(node(path))[permission] = new\_perm$.
	2	$path$ doesn't exist	HANDLE_PATH_NOT_EXIST($path$).
	22	$path$ is directory	HANDLE_PATH_EXIST($path$, DIR).