# Towards a Believable Decoy System: Replaying Network Activities from Real System

Jianhua Sun
College of William and Mary
jsun01@email.wm.edu

Kun Sun
George Mason University
ksun3@gmu.edu

Qi Li
Tsinghua University
qi.li@sz.tsinghua.edu.cn

***Abstract*—Recently cyber deception has emerged as a promising defense approach for detecting and defeating advanced persistent threat. By leveraging deceptive decoys, the defenders seek to proactively engage with the attackers and entice them away from the protected server infrastructure. The effectiveness of such decoy-based deception largely relies on the decoy fidelity. In this paper, we observe that realistic server system inevitably experiences wearoff from service request processing and regular maintenance, resulting in characteristic access pattern, running states, and system artifacts. Accordingly, we identify two deception evasion attacks, namely, *traffic fingerprinting* and *system fingerprinting*, which enable sophisticated adversaries to accurately distinguish decoys from real servers. To protect web server decoys against those evasion attacks, we develop Mirage, a seamless real-time network traffic replay framework to generate network traffic and system artifacts on the decoy server based on the normal clients' interactions with the real server. Mirage works as a TLS-capable reverse proxy that transparently replays real traffic towards decoys. To resolve the inconsistent states between the real and decoy servers, we integrate a decoy client emulator into the reverse proxy to maintain the stateful data features and caching logic of a decoy session. Moreover, we employ format preserving encryption technique to obfuscate sensitive data before being sent to the decoy server. Implementations and evaluations of a prototype demonstrate that Mirage can effectively mitigate deception evasion attacks with acceptable performance overhead.**

## I. Introduction

Sophisticated targeted attacks continue to be a severe threat against critical server infrastructure. In advanced persistent threat (APT) that exploits zero-day vulnerabilities or social engineering, well-resourced adversaries are able to circumvent current perimeter security systems (e.g., firewalls and intrusion detection systems) and gain stealthy access to the target system and its valuable information asset [1]. Even worse, attackers are increasingly employing DDoS attacks or fake ransomware to disguise the underlying targeted attack [2], further rendering traditional preventive security measures ineffective.

As a result, recently there has been a surging trend of leveraging *deception* to detect and defeat advanced persistent threat, where a number of *decoys* (or *traps*) are deployed throughout the protected network to lure intruders away from real assets [3]. By dedicating a trapped environment to engage with the attackers, a decoy can greatly expedite attack intelligence gathering [4], [5]. Compared to traditional IDS systems, decoy systems can report intrusions with nearly zero false positive rate, since normal users have no intentions to access those decoys. Moreover, decoys can even feed intruders with falsified information such as forged document, fake password and artificial OS fingerprint [6], [7], [8].

This work is partially supported by the U.S. ONR grant N00014-16-1-3214, ONR grant N00014-16-1-3216, and ONR grant N00014-18-2893.

Existing deception solutions mostly utilize *honeypots* to build up decoys [4], [9], [10]. Since low interaction honeypots can be easily identified using either timing or fingerprinting anti-honeypot attacks [11], they are not suitable for constructing decoys that aim at trapping persistent targeted attacks. Instead, decoys are usually created using high interaction honeypots or even authentic software duplicates of the real server, where the real system in one VM is cloned and deployed as a decoy VM [12]. Note that due to the proliferation of virtualization in cloud computing, it is not reliable for attackers to adopt anti-VM approaches [13] for the identification of virtual decoys.

In this paper, we first discover a critical limitation of existing high-fidelity decoy design - *the resulting decoys are statically deployed without any normal client interactions*. Given this observation, we propose two deception evasion attacks, *network traffic fingerprinting* attack and *system fingerprinting* attack. First, when attackers can passively eavesdrop the traffic within the decoy network and thus analyze the traffic pattern, they can successfully discriminate a decoy server from a real server, since decoys receive no service requests from real users. Second, after breaking in a host, the attacker can collect system artifacts closely related to its running states and access history and then leverage these artifacts to infer its genuineness. This is because a real server inevitably experiences wearoff due to continuous request processing and regular maintenance/upgrade, which do not happen on the decoy server. To verify the feasibility of these attacks, we implement a portable probe tool for collecting the characteristic artifacts and demonstrate the discriminatory power of these artifacts for accurate decoy identification.

To defend against those decoy evasion attacks, we develop a seamless real-time network traffic replay system called Mirage for automated decoy traffic and artifact generation. Since simply replaying real user requests to decoys can readily disrupt the decoy server states (e.g., different server IP addresses), we face several challenges when designing the seamless replay system that requires no changes on the real systems. First, stateful data features could be associated with an HTTP session, though HTTP is designed as a stateless protocol. Mishandling these features may result in discrepancy between the real and decoy server states. Second, widely implemented client-side caching and local storage may generate server-specific conditional requests. Third, simply replaying real user requests to a decoy server can possibly endanger user privacy once the decoy is compromised.

In response, Mirage operates as an TLS-capable reverse proxy that transparently intercepts encrypted HTTPS requests from real users and then replays them towards the decoys. To handle the stateful data features, it employs a *decoy client emulator* to emulate client interactions with the decoy server and maintain client-specific decoy server states for each HTTP session. The private caching logic of common browsers is integrated to avoid decoy state disruption caused by mishandling of conditional requests. To protect user privacy during the replay process, Mirage uses format-preserving encryption technique to obfuscate sensitive user data. By deploying the proxy close to the server side, Mirage is scalable to handle concurrent client requests without requiring any client-side changes. We implement a Mirage prototype and the experimental results show that Mirage can effectively mitigate decoy evasion attacks with acceptable performance overhead.

In summary, we make the following contributions:

- We discover a critical limitation of existing decoy designs due to its lack of normal client interactions and identify a variety of system artifacts characterizing the running states and access history of a real server system.
- We propose two deception evasion attacks that allow sophisticated adversaries to circumvent decoy-based deception by analyzing online server access pattern and system artifacts. We develop a proof-of-concept prototype of the identified attacks to verify its feasibility on decoy evasion.
- As a countermeasure, we develop a seamless real-time replay framework for automatically generating the network traffic and system artifact on the decoy web server. Our reverse proxy-based system prototype shows its effectiveness on defeating deception evasion attacks with small performance overhead.

## II. Threat Model and Assumptions

We consider targeted attacks (e.g., advanced persistent threat) against enterprise servers and networks. We assume the enterprise network consists of real servers and decoy servers, where the decoy servers are installed with the same system and software stack as the real servers in order to misinform the attackers. Due to the prevalence of web-based interface for remote access, we focus on studying the attacks and protection of web server decoys.

We assume the attackers may know the targeted enterprise networks are protected with decoy servers. Therefore, attackers should accurately distinguish decoys from real servers to avoid exposing the valuable attack vectors and being misinformed with false data. The attacker can launch both active and passive attacks. It can first compromise a host in the network and actively examine its system artifacts. Furthermore, the attacker can sniff the local network traffic and analyze the network access patterns. Our defensive goal is to mislead the attackers so that they can not perceive if the compromised host is real or decoy. Thus, even when multiple hosts (including the real server) are compromised, the attacker still cannot recognize which copy of sensitive data is genuine.

## III. Deception Evasion Attacks

Existing deception systems typically employ static decoys to entice the attackers away from the protected servers. However, we identify a critical limitation of existing decoy designs: *lacking normal client interactions*, which leads to disparate decoy access pattern and system states. Given this, we present two attacking techniques that allow sophisticated adversaries to circumvent deception: *network traffic fingerprinting* and *system fingerprinting*.

### A. Network Traffic Fingerprinting

We first envision the possibility of invalidating deception through simple network traffic fingerprinting. Once penetrating a system, insider attackers can distinguish the decoys from the real servers by eavesdropping (e.g., through ARP spoofing) the traffic and analyzing the host access pattern. Due to the lack of real user access, standalone decoys can be easily sifted out.

Statistical traffic analysis [14] refers to the set of techniques that leverage flow information (e.g., packet sizes and timing) for traffic pattern classification. Inspired by its application to website fingerprinting [15], we identify a diverse set of features that characterize a typical server system. These features allow the attacker to infer decoy identity based on its traffic pattern. We consider the exploitable features in the following:

- General features, which include total transmission size, duration, and the number of inbound/outbound packets/bytes.
- Aggregate features, which include the overall distribution of packet lengths and inter-arrival times.
- Unique packet lengths. This feature vector is constructed where a feature element is defined as 1 if the corresponding length value occurs and 0 if it does not.
- Packet size frequency. This feature is an augmentation of the unique packet length feature, which is represented by a multiset $l = l_1^{f_{l_1}}, l_2^{f_{l_2}}, \cdots, l_m^{f_{l_m}}$ containing packet length $l_j$ and the respective frequency $f_{l_j}$.
- Packet ordering. We define an *inter-arrival time trace*, $\mathtt{I}$, as the sequence of $n$ inter-arrival times, $\{t_1, t_2, \cdots, t_n\}$, between packets for a trace. The trace associated with a server is collected during a fixed time frame (e.g., per day). Similarly, we also define a *packet size trace*, as the sequence of $n$ packet sizes, $\{l_1, l_2, \cdots, l_n\}$.

### B. System Fingerprinting

Motivated by the malware sandbox evasion technique utilizing "wear and tear" artifacts [16], we argue that a genuine server also generates system artifacts due to normal client interactions and regular server maintenance/upgrade. These artifacts compose a robust indicator of a real server's actual access history. However, we focus on the identification of decoy server instead of end-user sandbox, and discover a disparate set of novel artifacts. We also include artifacts profiling the states of the server system, e.g., performance metrics of various server subsystems. Table III summarizes the artifacts we identified for a typical LAMP web server stack.

*1) System:* Considering that production servers typically remain accessible after boot-up except infrequent failures, several artifacts are generated during server usage. General system properties such as the number of created processes/threads are directly correlated to the server access history. Other system metrics including context switches, serviced interrupts/softirqs and page faults also get accumulated from user request processing. For example, in Linux kernel each packet reception triggers a hardware interrupt in which context the network driver raises a softirq for further packet processing.

Similarly, routine server administration involves necessary system upgrade, application update or security patching. These update related events are archived for debugging and error forensics. Another category of artifacts is related to fine-grained CPU time allocation, which provides characteristic profiling of server workload. Decoy server remains idle during its life cycle, while a busy web server may dedicate tremendous amount of time handling interrupts or waiting for disk I/O. Therefore, we incorporate features related to CPU usage, e.g, the percentage of CPU time spent on executing user processes, waiting for I/O and servicing interrupts, etc.

*2) Disk and File System:* User interactions inevitably result in file system changes, especially for I/O intensive operations (e.g., database transactions). Critical server configuration and resource files, e.g., the Apache configurations and website resources, are not susceptible to attacker scrutinization since it requires escalated user privileges. Therefore, we limit the disk artifacts within the normal user readable files. This embraces system and application generated temporary files, files cached in memory and application data files cached on disk. We also include the numbers of completed disk read/write operations and the time consumed by disk I/O activities.

*3) Network:* The networking system is a valuable source of exploitable artifacts. The number of ARP cache entries indicates the connectivity environment. The number of active connections reveals the server's state as decoys possess occasionally active connections. As a indicator of history, we record the accumulated numbers of passively and actively opened TCP connections as well as transmitted and received packets. Besides, the number of application related server processes and threads (e.g, the Apache and MySQL server processes) reveals the concurrency of user requests.

*4) System Logs:* The server logging system serves as another rich repository of artifacts. It contains vast amount of information about the server's operations, providing detailed insights of its performance, security and underlying issues. Due to constrained user privilege, our data collection is restricted to user readable log files. In particular, the `update-alternatives` logs record changes of links to default program binaries. The system update utility such as `Apt` and `dpkg` are two important artifact sources. We calculate the total size of the corresponding logs, and the elapsed time between the oldest and newest update event. We also incorporate generic system activity logs storing informational and non-critical system messages, kernel ring buffer messages, system authentication history and users' login/logout history. Application logs (e.g., the Apache web server access and error logs) are usually inaccessible to normal users without `admin` privilege and are out of the scope of our analysis.

*5) System Performance Metrics:* System performance metrics (e.g., CPU, memory and I/O) compose characteristic profile of server activity. The rate of process creation and context switch depicts server state. The size of runqueue, task list and the average load collectively describes the real-time workload. For example, the Apache web server spawns subprocesses or threads to cope with changing workload. Likewise, the memory subsystem exhibits distinct artifacts (e.g., the respective size of free, used and cached memory) that reveal the inner working of the server.

Similarly, we monitor I/O-related metrics including the rate and throughput of read/write disk transactions, the amount of free/used swap space, and network performance stats (e.g., average packet transmission/receiving rate and the number of created and in-use sockets).

## *C. Instantiating Fingerprinting Attack*

Network traffic fingerprinting is trivial to implement provided that the attacker is capable of sniffing decoy traffic. For system fingerprinting, we implemented a artifact collection tool based on POSIX API. It consists of 1300 lines of C code and 120 lines of bash script. Although it is tailored for the LAMP web server stack, it is readily portable to most UNIX/Linux OSes as it requires no extra dependencies besides already available POSIX interface.

To demonstrate the discriminatory power of the artifacts, we conceive a controlled setup consisting of the web server of a university lab and a decoy server. Since decoys with vanilla OS are susceptible to detection, we directly replicate the system image of the real server so that the decoy's initial configuration is believable. After that the decoy remains idle, while the real server continuously processes service requests. Specifically, we record a real user's trace of visiting real server for an hour using *mitmproxy* [17], and then generate 100 virtual users to replay the recorded trace for a week, during which we used our tool to collect the artifacts from both systems.

Figure 1 shows the normalized value of each decoy artifact compared to the real server artifact. We can identify the divergence as the majority of artifacts are accumulated in real server during replay except the `idleCPUPercentage` and `freeMemorySize`. To quantify the prominence of each artifact, we apply linear regression to the artifact vector and

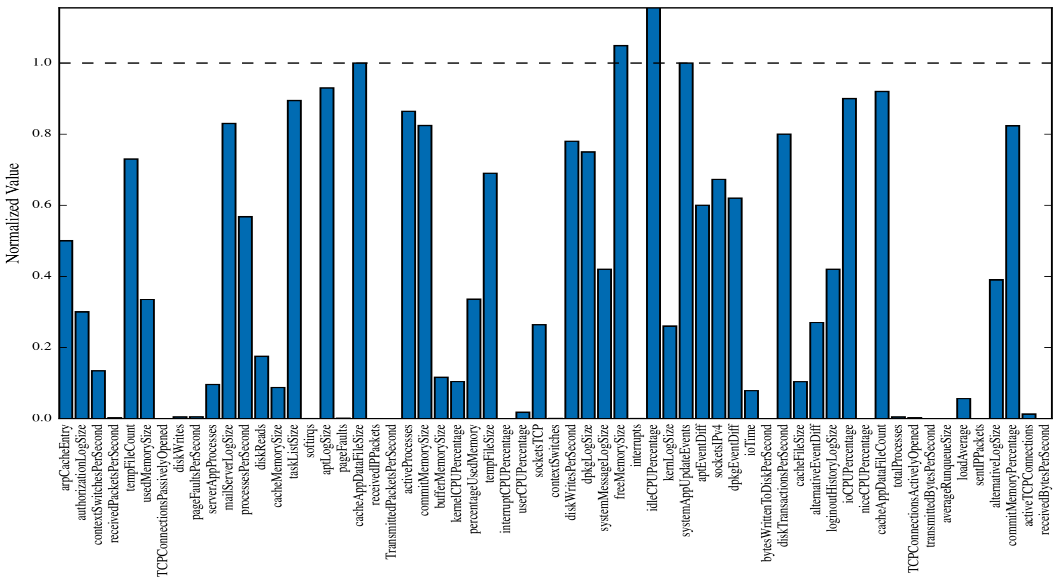

Fig. 1: Normalized value of decoy artifacts relative to real server artifacts

calculate the trend of accumulation. Table I lists the top 15 most significant artifacts. We can see that continuous user request processing results in fast accumulation of artifacts such as page faults, context switches and interrupts, etc. Therefore, normal user interactions and genuine server usage artifacts are essential to constructing high-fidelity decoys, and the fidelity of a decoy being a real server to the attackers hinges on its capability of emulating real server usage pattern and history.

TABLE I: Top 15 most significant artifacts

| Artifact | Accumulation Rate (per hour) |
|---|---|
| pageFaults | 44M |
| contextSwitches | 36.7M |
| softirqs | 23.6M |
| interrupts | 20.3M |
| receivedIPPackets | 6.5M |
| sentIPPackets | 3.8M |
| TCPConnectionsPassivelyOpened | 0.3M |
| systemMessageLog | 0.2MB |
| usedMemory | 77.8MB |
| authorizationLog | 0.08MB |
| kernLog | 0.06MB |
| diskWrites | 0.04M |
| totalProcesses | 0.01M |
| bufferMemory | 2.5MB |
| TCPConnectionsActivelyOpened | 1.8K |

## IV. Mirage Design

To defeat the identified deception evasion attacks with authentic decoy access pattern, we develop a real-time web replay system Mirage. In the following, we first give a system overview and then elaborate on how Mirage resolves the challenges on achieving seamless web replay.

### A. System Overview

Figure 2 shows the system architecture of Mirage. Each real server is accompanied by at least one decoy server. To replay real traffic to decoy, a replay engine is deployed at the network edge working as a TLS-capable reverse proxy that serves resources on behalf of the real server. Specifically, it establishes a secure TLS connection with the client by impersonating the real server, and negotiates two separate TLS connections to the real and decoy servers. In this way, it intercepts and decrypts a client request, duplicates it to generate decoy request, and randomly replays both requests to the real server and decoy, respectively. The real server's response is delivered to the client through the same encrypted connection; while the decoy response is digested by the decoy client emulator.

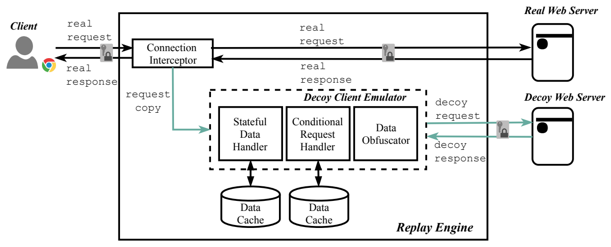

Fig. 2: Mirage system architecture

### B. Seamless Web Replay

After scrutinizing the HTTP protocol and dominant web server architectures, we identify various HTTP features that are critical for seamless replay. Since client state directly correlates with server response, the replay engine leverages a *Decoy Client Emulator* to emulate the state transitions of a decoy client interacting with the decoy. It tracks decoy responses and recognizes any discrepancy needed to be handled for replay.

**Emulating stateful HTTP behaviors.** The stateless HTTP(S) protocol can be forced to be stateful in real-world deployment. The server can first send the state representation to the client and expect to receive it in subsequent requests. Overall there are three ways to achieve this in HTTP: cookies, URL rewriting and hidden form fields, where the state is embedded in HTTP header, URL and hidden form, respectively.

Mishandling of these stateful features can result in divergent real and decoy server states. Figure 3 illustrates server state divergence caused by misconfigured cookies. Initially, decoy server injects a different or extra baiting cookie into its response (e.g., different SessionID for session tracking). This cookie is nonexistent in the real server response or has different value/attribute. Naively replaying subsequent real requests to decoys almost always triggers distinct decoy response. For example, the decoy may abort the session to prevent session hijacking attack since the provided session cookie is invalid.

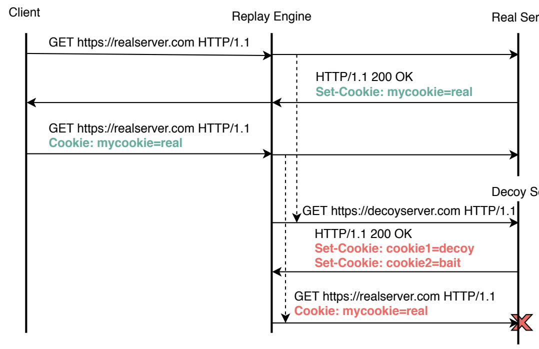
Client
Replay Engine
Real Server
GET https://realserver.com HTTP/1.1
HTTP/1.1 200 OK
Set-Cookie: mycookie=real
GET https://realserver.com HTTP/1.1
Cookie: mycookie=real
Decoy Server
GET https://decoyserver.com HTTP/1.1
HTTP/1.1 200 OK
Set-Cookie: cookie1=decoy
Set-Cookie: cookie2=bait
GET https://realserver.com HTTP/1.1
Cookie: mycookie=real

Fig. 3: Divergent server states due to mishandling of cookies

To maintain decoy state, the *Decoy Client Emulator* includes a *Stateful Data Handler* to handle the stateful features. For each user session, it emulates the interaction between a decoy client and the decoy server. As shown in Algorithm 1, for each session it intercepts and analyzes the responses to extract any stateful decoy data. If so, the stateful data is temporarily cached locally and indexed by the client address and request URI. The metadata (e.g., cookie attributes such as expiration time, valid domain, path, etc.) are also stored along with the data name and value. Whenever subsequent client requests involves such stateful data, the stateful data handler first checks whether there exist corresponding decoy data entries in the local store. If that is the case, it retrieves the data records including the associated data attributes. After parsing the data attributes and verifying its validity, the replay engine then replaces real data with decoy data in the replayed request.

**Resolving inconsistency from conditional requests.** HTTP cache validation uses conditional requests to check if a resource matches the version indicated by a *validator*. Validators are specified in conditional headers (e.g., `If-None-Match` and `If-Modified-Since`), including its `last-modified` date and an opaque hash string (i.e., *entity tag* or `etag`). For replay, conditional requests may cause inconsistent server responses. As shown in Figure 4, initially the client fetches a file via a `GET` request and also replays it to the decoy. Upon receiving the requests, both servers instruct the usage of caching in a `Cache-Control` header and attach a cryptograph hash-based `ETag` (e.g., MD5 hash) to mark the resource. However, decoy may generate a different ETag hash or set different cache expiration time. Whenever the client issues conditional request to check whether the cache is stale, the real server may return a "`304 Not Modified`" message without sending the resource. However, the replayed request may elicit a complete decoy response since the enclosed ETag fails the validity check.

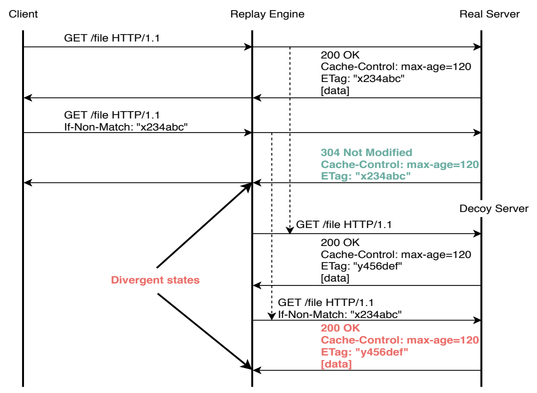

Fig. 4: Inconsistency caused by conditional request

To guarantee consistent decoy response, the Decoy Client Emulator emulates the client's private caching behavior through a *Conditional Request Handler*. As shown in Algorithm 1, when caching is enabled, a per-client hash table entry is created mapping the accessed resource with the validators and caching directives (e.g., `max-age` and `expires`). For ensuing client conditional request, depending on the precondition headers, it performs the cache validation to determine whether the conditional request should be replayed to the decoy. First, it calculates the cache expiration time based on the response time, the freshness lifetime and current age of cache. If the cache becomes stale, it replays the conditional request to the decoy with the precondition header fields replaced by the proper decoy values. Otherwise, no decoy request is replayed. In the case that only ETag-based validation is enabled, the request still gets replayed with the ETag modified as the most recent value set by the decoy.

**Obfuscating sensitive user data.** Sensitive user data is often involved in HTTP transactions such as user credentials and authentication tokens (e.g., JSon web token). Inadvertently replaying unsafe state-changing HTTP requests (i.e., `PUT` and `POST`) can leak sensitive user data once a decoy is compromised. Thus, it is imperative to devise a sensible strategy for inline data obfuscation. However, with unpredictable ciphertext format, obfuscation using typical encryption schemes can readily disrupt decoy server state due to failures of server-side data format validation. Therefore, we use a data obfuscation scheme that can preserve both data format and confidentiality.

Specifically, *Data Obfuscator* extracts user data from the requests and applies *format-preserving encryption* (FPE) [18] to generate obfuscated copies. It processes user data submitted as URL encoded form using `GET` and multi-part form using `PUT`/`POST`. To achieve FPE on data records containing strings of arbitrary length, the "rank-encipher-unrank" approach is adopted. For a string consisting of $N$ characters [A-Za-z], it is treated as a base-52 value with each character being a

**Algorithm 1** Seamless web replay

```
global $stateful\_cache \leftarrow$ hashtable()
global $data\_cache \leftarrow$ hashtable()
procedure HANDLE_REQUEST($flow$)
    if $flow.request$ and $flow.request.dst == real$ then
        $request \leftarrow flow.request$
        $decoyrequest \leftarrow flow.request$
        if is_stateful($request$) then
            $key \leftarrow$ state_key($flow$)
            $decoy\_state \leftarrow stateful\_cache$.get($key$)
            for $eachstate \in decoy\_state$ do
                if is_valid($eachstate$) then
                    modify_state($decoyrequest$, $eachstate$)
                end if
            end for
        end if
        if is_conditional($decoyrequest$) then
            $key \leftarrow$ cache_key($flow$)
            $validator, directive \leftarrow data\_cache$.get($key$)
            if not $validator$ then
                $data\_cache$.remove($key$)
            else if expired($directive$, $validator$) then
                modify_cheader($decoyrequest$, $validator$)
            end if
        end if
        if $formdata$ in $decoyrequest$ then
            for $eachfield \in formdata$ do
                $eachfield \leftarrow$ fpe($eachfield$)
            end for
        end if
        replay($decoyrequest$)
    end if
end procedure
procedure HANDLE_RESPONSE($flow$)
    if $flow.response$ and $flow.response.source == decoy$ then
        $response \leftarrow flow.response$
        if is_stateful($response$) then
            $key \leftarrow$ state_key($flow$)
            $stateful\_data \leftarrow$ extract_data($response$)
            $stateful\_cache[key] \leftarrow stateful\_data$
        end if
        if is_cacheable($response$) then
            $directive \leftarrow$ parse_cache_control($flow$)
            $key \leftarrow$ cache_key($flow$)
            if cache_enabled($directive$) then
                $validators \leftarrow$ parse_validators($response$)
                $data\_cache[key] \leftarrow directives+validators$
            else
                $data\_cache$.remove($response$)
            end if
        end if
    end if
end procedure
```

digit (i.e., A=0, B=1, ..., y=50, z=51). Then we do a base conversion of the value to an integer between 0 and $51^N - 1$. After that we use a standard integer FPE technique to encrypt this value into another integer in the same range, and do a base conversion back into a new string of $N$ characters.

For data that must abide by a format specified as regular expression, the replay engine uses a FPE variation scheme `libfte` [19] that represents the regex as nondeterministic finite-automata (NFA) and performs relaxed ranking on the NFAs. By analyzing the hosted web application, the replay engine acquires the formats of the data fields (e.g., in user-submitted web form) in advance and generates regular expressions for each field. It then transforms each data field into an obfuscated version before replaying the request.

## V. Prototype Implementation

We implement a Mirage prototype based on *mitmproxy* (version 2.0.2) [17]. The implementation comprises $\sim$ 1050 lines of python script. Its reverse-proxy mode is used to transparently proxify the real/decoy servers. The real-time replay scheme is implemented using its inline scripting interface controlled and executed by the flow master *FlowMaster*. *mitmproxy* establishes an event-based mechanism where various events are raised and transferred through a dedicated control channel to trigger corresponding handler routines of the FlowMaster. A `clientconnect` event is raised whenever a client initiates a connection, and a `request/response` event is raised once a client request or server response are received.

Specifically, an HTTP transaction is abstracted by an `HTTPFlow` class containing a collection of child objects representing HTTP request (`HTTPRequest`), response (`HTTPResponse`) and errors (`HTTPError`). We programmed handler routines for `clientconnect`, `clientdisconnect`, `request` and `response` events. Each handler consumes a `HTTPFlow` object. Hash tables are used to temporarily store decoy data. Client information is extracted to construct index key once `clientconnect` event is raised. A `clientdisconnect` event invokes a procedure to clean up useless hash table entry (e.g., session cookie or expired cache entry). The identification and extraction of stateful data features and cacheable data are implemented in the `response` handler. The stateful feature replacement, caching logic, and data obfuscation are programmed in the `request` handler. To implement client private caching, only caching-related headers and directives are stored to reduce memory footprint. We use *Beautiful Soup* [20] to parse and extract hidden form fields. The libFTE library is adapted and integrated into the replay system to perform format-preserving encryption on form data.

## VI. Evaluation

We evaluate the effectiveness of Mirage in mitigating traffic fingerprinting and system fingerprinting attacks and investigate performance of the implemented replay scheme. All experiments were performed on a desktop with Intel(R) Xeon(R) E5-2620 CPU and 16GB RAM running 64-bit Ubuntu 16.04. We use VirtualBox to create a real server VM, a decoy VM, a proxy VM and a test VM, all running Ubuntu 16.04 with 1GB memory. To test replay in a controllable environment, we develop a customized dynamic Django web application supporting CRUD (i.e., `create`, `read`, `update`, and `delete`) operations. The application provides a library service for users to look up book inventory, create/update personal profile, borrow and renew book after registration. Real server VM and decoy VM are installed with Apache HTTP server 2.4.18 which serves the web application through WSGI interface. The proxy VM runs an instance of Mirage, forwarding requests to the real/decoy server VMs. From test VM we emulate real users requesting services from the real server via proxy VM.

### A. Deception Evasion Attack Mitigation

To assess the effectiveness of replay, from test VM we continuously access the real server for an hour and record the inbound/outbound traffic at the real server and decoy VMs.

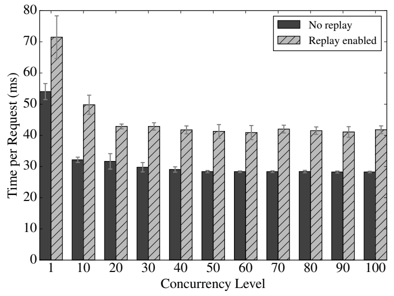

Fig. 5: Round-trip time

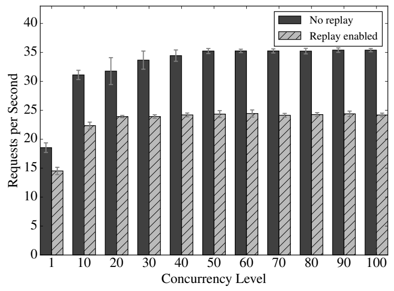

Fig. 6: Throughput

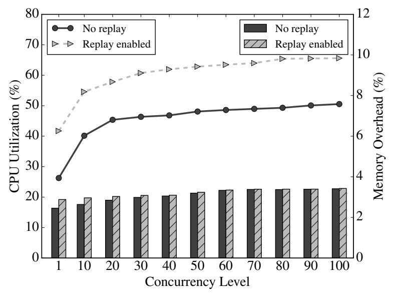

Fig. 7: System overhead

TABLE II: Comparison of real and decoy server access features

| Server Category | Transmission Size(MB) | Duration(s) | Average Packet Size(B) | Number of Inbound Packets | Number of Outbound Packets | Packet Size Trace Similarity | Inter-arrival Time Trace Similarity |
|---|---|---|---|---|---|---|---|
| real server | 2.01 | 3585.72 | 240.5 | 4120 | 4201 | 1.00 | 0.99 |
| decoy server | 2.01 | 3585.72 | 240.5 | 4116 | 4215 | | |

**Server access pattern.** We first analyze the similarity of aggregate features. Figure 8 and Figure 9 show the histograms representing the distributions of packet sizes and inter-arrival times. Decoy server exhibits feature distributions completely resembling real server. Regarding the overall traffic profile, the total transmission size, duration, packet counts and ordering features are listed in Table II. To compare the inter-arrival time trace and packet size trace, we compute the cosine similarity between two traces. The result indicates that Mirage can seamlessly generate decoy access pattern indistinguishable from the real server.

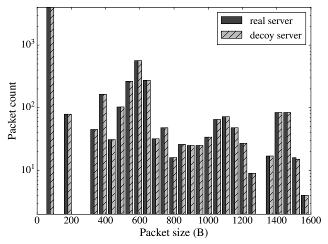

Fig. 8: Packet size distribution

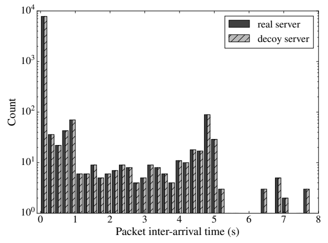

Fig. 9: Interarrival time distribution

**System fingerprinting.** To show the usefulness of replay for artifact generation, we boot a server VM, a decoy VM and another decoy VM clone from the pristine state. Then from the test VM, we randomly replay the recorded trace for two weeks with automatic daily update enabled for the server VM and decoy VM. In particular, the proxy replays all requests to the decoy VM while the clone VM remains idle. We then collect the artifacts accumulated in the three VMs using our probe tool and list the values in Table III. We can see that Mirage can generate decoy artifacts comparable to the real server, while the idle decoy only possesses fairly limited artifacts.

## B. Performance Overhead

**Impact on legitimate connections.** We first estimate the impact of replay on normal user connections. We use `curl` to continuously send $1000$ requests to the real server and measured the round-trip times. Enabling replay incurs on average $12ms$ extra latency for normal user connection. Considering that it is normal for web server to serve request through a proxy, this extra latency is usually tolerable for users and does not deteriorate the web application's responsiveness [21].

**Replay time breakdown.** To assess the responsiveness of the replay system to diverse requests, we measured the processing time breakdown for stateful feature handling, caching control and data obfuscation. In this experiment, `GET` and `POST` requests are generated using Firefox to trigger different processing functions. The interaction is sketched as follows: $(1)$ user connects to the website main page; $(2)$ server returns a web form requesting for user credential (i.e., username and password); $(3)$ server returns the main page to the user and sets cookies to maintain user session and validators for cache control; $(4)$ user uploads personal information, resulting in a HTML form from the server with hidden csrf token embedded to protect against cross-site request forgery. Since our replay system is event-based and a function may involve two-phase processing whenever either a request or response is received, we measured the time separately. The results are given in Table IV. Overall, caching control accounts for the majority of the processing time due to the complexity of the implemented caching logic and the volume of cached data. Handling stateful data features incurs far less latency overhead since it only requires simple header substitution. The overhead for data obfuscation is directly correlated with the data volume.

TABLE IV: Processing time breakdown

| Function | Request (ms) | Response (ms) |
|---|---|---|
| stateful feature handling | 0.63 | 0.15 |
| caching control | 2.85 | 6.02 |
| user data obfuscation | 0.46 | 0.06 |
| request duplication | 2.97 | n/a |
| response parsing | n/a | 9.15 |

**Scalability.** To test the scalability of Mirage, we used *ab* to create a massive workload of legitimate users ($5000$ requests in up to $100$ concurrent threads) from the test VM. Figure 5 and Figure 6 show the round-trip time and throughput overhead with respect to concurrency. The replay system can handle strenuous workload with consistent overhead (i.e., on average $12.5ms$ in RTT and $10.4$ $requests\, per\, second$ in throughput) even when replaying multiple sessions concurrently.

**Overall system overhead.** We also measured the overall system overhead incurred by replay. To simulate different

TABLE III: System artifacts

| Category | Name | Description | Real Server | Decoy with Replay | Idle Decoy |
|---|---|---|---|---|---|
| | totalProcesses | # created processes and threads since boot | 4679 | 4603 | 2031 |
| | totalActiveProcesses | # actively running processes | 163 | 165 | 147 |
| | sysupdt | # system and application update events | 21 | 19 | 0 |
| | ctxt | # context switches across all CPUs | 12.68M | 11.95M | 2.73M |
| | totalIrq | # serviced interrupts since last boot | 5.73M | 5.58M | 0.12M |
| | softirqs | # serviced softirqs since last boot | 5.62M | 5.44M | 0.46M |
| | totalPageFaults | # page faults since last boot | 4.35M | 4.32M | 0.23M |
| | userCPUPercentage | % CPU time running normal processes | 0.66 | 0.65 | 0.07 |
| | kernCPUPercentage | % CPU time executing kernel | 0.53 | 0.51 | 0.03 |
| | idleCPUPercentage | % CPU time in idle state | 96.28 | 96.19 | 99.25 |
| | ioCPUPercentage | % CPU time waiting for I/O to complete | 0.52 | 0.48 | 0.69 |
| | irqCPUPercentage | % CPU time servicing interrupts | 0.05 | 0.05 | 0 |
| General | procPerSec | # processes created per second | 1.92 | 1.87 | 0.06 |
| | ctxtPerSec | # context switches generated per second | 215.21 | 190.84 | 50.85 |
| | pageFaultsPerSec | # page faults triggered per second | 177.34 | 196.57 | 0.34 |
| | sizeRunqueue | Average size of Linux system run queue | 3 | 2 | 0 |
| | sizeProcList | Average size of task list | 521 | 510 | 407 |
| | loadAvg | Load average during a certain past duration | 1.64 | 1.72 | 0.18 |
| | sizeMemFree | Total size of free system memory (Bytes) | 668.38M | 665.57M | 957.31M |
| | sizeMemUsed | Total size of used system memory (Bytes) | 1.37G | 1.38G | 1.08G |
| | PercentageMemUsed | % used memory with respect to total memory | 86.96 | 82.7 | 40.35 |
| | sizeMemBuffer | Amount of memory used as buffer by kernel (Bytes) | 105.63M | 104.22M | 33.6M |
| | sizeMemCached | Amount of memory used to cache data by kernel (Bytes) | 570.62M | 556.31M | 356.74M |
| | sizeMemCommit | Amount of memory needed for current workload (Bytes) | 3.59G | 3.41G | 1.73G |
| | percentageMemCommit | % memory needed for current workload | 86.96 | 82.7 | 40.35 |
| | tempFilesSize | Size of temporary system files (Bytes) | 632K | 568K | 46 |
| | tempFilesCount | # temporary system files | 23 | 22 | 2 |
| | cacheFilesSize | Size of cached system and application files (Bytes) | 1213M | 1217M | 68M |
| | dataCacheFilesSize | Size of cached application data on disk (Bytes) | 114.8M | 115.1M | 4.68M |
| | dataCacheFilesCount | # cached application data files on disk | 247 | 247 | 19 |
| | diskReadsCount | # completed disk reads | 30672 | 30576 | 2953 |
| Disk | diskWritesCount | # completed disk writes | 19175 | 18563 | 2595 |
| | timeIO | Total time spent doing I/O (seconds) | 323 | 298 | 2.04 |
| | transactionPerSec | # disk transactions per second | 4.23 | 3.76 | 0 |
| | readTransactionPerSec | # disk read transactions per second | 1.35 | 1.07 | 0 |
| | writeTransactionPerSec | # disk write transactions per second | 2.91 | 2.77 | 0 |
| | readBytesPerSec | Bytes read per second | 93.87 | 90.72 | 0 |
| | writeBytesPerSec | Bytes written per second | 128.3 | 122.67 | 0 |
| | ARPCacheEntries | # entries in ARP cache | 3 | 3 | 1 |
| | tcpConnections | # active TCP connections | 14 | 13 | 1 |
| | activeTCPConnections | # actively opened TCP connections since boot | 17 | 16 | 1 |
| | passiveTCPConnections | # passively opened TCP connections since boot | 685.08K | 685.04K | 1 |
| | pktTransmitCount | # transmitted packets since last boot | 378.25M | 363.76M | 412 |
| | pktRecvCount | # received packets since last boot | 3.43G | 3.42G | 536 |
| Network | totalAppProcesses | # of live application server processes | 12 | 12 | 4 |
| | totalSockets | # of total sockets in use | 1384 | 1164 | 599 |
| | totalTCPSockets | # of TCP sockets in use | 87 | 79 | 2 |
| | rxPktsPerSec | # received packets per second | 507 | 485 | 0 |
| | rxBytesPerSec | # received bytes per second | 42.85 | 40.56 | 0 |
| | txPktsPerSec | # transmitted packets per second | 8783 | 8704 | 0 |
| | txBytesPerSec | # transmitted bytes per second | 985.49 | 969.36 | 0 |
| | updateAlternativesLogSize | Size of update alternatives logs (Bytes) | 4.41K | 4.53K | 0.31K |
| | diffUpdateAlternativesLog | Time between the oldest and newest update-alternatives event (days) | 14 | 14 | 1 |
| | aptLogSize | Total size of Apt logs (Bytes) | 33.93K | 34.17K | 1.04K |
| | diffAptLog | Time between the oldest and newest Apt update event (days) | 14 | 14 | 1 |
| | dpkgLogSize | Total size of Dpkg logs (Bytes) | 131.36K | 128.29K | 16.9K |
| Server Logs | diffDpkgLog | Time between the oldest and newest Dpkg update event (days) | 13 | 14 | 1 |
| | authLogSize | Total size of system authorization logs (Bytes) | 87.45K | 63.07K | 7.21K |
| | sysLogSize | Total size of global system message logs (Bytes) | 359.67K | 423.62K | 42.57K |
| | kernLogSize | Total size of kernel logs (Bytes) | 575.39K | 593.22K | 90.18K |
| | mailLogSize | Total size of mail server logs (Bytes) | 218.22K | 196.32K | 7.32K |
| | wtmpLogSize | Total size of login records logs (Bytes) | 39.52K | 31.48K | 3.72K |

workloads, we used *ab* to stress test the replay system for an hour with at most 100 concurrent user threads. We use `pidstat` from `sysstat` [22] to capture statistics about CPU usage and `pmap` to measure memory consumption. As shown in Figure 7, replay results in 9.2% increase in CPU utilization and negligible memory consumption. Therefore, our system can be readily integrated into real-world web architectures without excessive performance overhead.

## VII. Discussion

Our replay design identifies remote client using network address. However, an edge gateway router usually serves as a network address translator (NAT), rewriting private IP addresses of individual devices to a single public IP assigned by the ISP. To accurately identify clients behind a NAT, various client identification mechanisms can be integrated into our replay system. We can leverage explicitly assigned client-side identifiers (e.g., HTTP cookie), inherent client device characteristics, or measurable user behaviors and preferences [23].

To create decoys with believable artifacts, another method is to clone real server directly. However, this strategy cannot generate real access traffic, rendering the artifacts obsolete eventually. We could also start with a pristine decoy system and simulate user interactions. This requires cumbersome client instrumentation, making it difficult to scale to multiple clients. In contrast, our solution can automatically generate system artifacts through scalable server-side traffic replay.

## VIII. Related Work

**Honeypots and deception.** Honeypots are network decoys conceived for attack detection and information gathering [9], [10]. By leveraging virtualization, virtual decoys can be introduced into the protected network to achieve high visibility into attacker activities [4], [5]. To mitigate large scale attack, hybrid honeypot systems (i.e., *honeyfarms*) [12] are developed to balance decoy fidelity and scalability. Using an anomaly detector, Shadow honeypots [24] forwards suspicious traffic to an instrumented shadow copy of target application and filter subsequent attack instances. Desir [25] employs deception to defeat the reconnaissance of remote targeted attacks. State-of-the-art deception also misdirects and disinforms attackers with falsified information, e.g., false passwords [26], artificial OS fingerprint [8] and fake document [6]. Even though these designs endeavour to create high-fidelity decoy environment, we discover their limitation and instead focus on the generation of believable decoy access pattern and history.

**Anti-decoy techniques.** Remote attackers can bypass virtual decoys through either timing or fingerprinting analysis [11]. Evasive malware can leverage virtualization artifacts (e.g. VM-specific device drivers) to identify decoy environment [27]. However, these anti-decoy techniques are not reliable due to the proliferation of virtualization in cloud computing. In [16], "wear-and-tear" artifacts are used to fingerprint malware sandbox. Our work adopts an analogous strategy for artifact identification. However, we focus on decoy server systems embracing significantly diverse and novel artifacts instead of end-user sandboxes.

**Record-and-replay frameworks.** Record-and-replay tools are typically used for network performance benchmarking and debugging. Web-page-replay [28] uses DNS indirection to intercept HTTP traffic during record and replay. Fiddler [29] adjusts system-wide proxy setting for traffic interception. With these tools, all HTTP requests and responses are recorded while passing through the proxy server. Mahimahi [30] improves upon prior tools by emulating the multi-server nature of web applications. WaRR [31] achieves reproducible application debugging by capturing and replaying program executions. All these tools either require cooperation between traffic source and sink or fail to reproduce the original timing property. In contrast, our seamless replay scheme can automatically generate decoy traffic and artifact in real-time.

## IX. Conclusion

This paper discovered the limitation of existing deception system designs due to the lack of normal client interactions. Given this limitation, we proposed and instantiated two decoy evasion attacks, namely, network traffic fingerprinting and system fingerprinting attacks, which allow sophisticated adversaries to circumvent existing decoy-based deception. We developed a seamless real-time replay framework as a countermeasure to defeat identified evasion attacks. The experimental evaluations of our system prototype demonstrate that our approach can effectively mitigate the decoy evasion attacks.

## References

[1] Symantec, "2018 Internet Security Threat Report," https://www.symantec.com/security-center/threat-report, 2018.

[2] Kaspersky Lab, "Kaspersky Lab identifies ransomware actors focusing on targeted attacks against businesses," https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-identifies-ransomware-actors-focusing-on-targeted-attacks-against-businesses, 2017.

[3] Drew Robb, "Deceiving the Deceivers: Deception Technology Emerges as an IT Security Defense Strategy," https://www.esecurityplanet.com/network-security/deception-technology.html, 2017.

[4] M. Beham, M. Vlad, and H. P. Reiser, "Intrusion detection and honeypots in nested virtualization environments," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013.

[6] 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 - 5th International ICST Conference, SecureComm 2009, Athens, Greece, September 14-18, 2009, Revised Selected Papers*, 2009, pp. 51–70.

[5] D. Dagon, X. Qin, G. Gu, W. Lee, J. B. Grizzard, J. G. Levine, and H. L. Owen, "Honeystat: Local worm detection using honeypots," in *Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings*, 2004, pp. 39–58.

[7] J. Voris, J. Jermyn, N. Boggs, and S. Stolfo, "Fox in the trap: Thwarting masqueraders via automated decoy document deployment," in *Proceedings of the Eighth European Workshop on System Security*, 2015.

[8] M. Albanese, E. Battista, and S. Jajodia, "A deception based approach for defeating os and service fingerprinting," in *Communications and Network Security (CNS), 2015 IEEE Conference on*, 2015.

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

[10] N. Provos and T. Holz, *Virtual Honeypots - From Botnet Tracking to Intrusion Detection*. Addison-Wesley, 2008.

[11] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, 2005, pp. 29–36.

[12] J. Sun, K. Sun, and Q. Li, "Cybermoat: Camouflaging critical server infrastructures with large scale decoy farms," in *2017 IEEE Conference on Communications and Network Security (CNS)*, 2017.

[13] P. Ferrie, "Attacks on virtual machine emulators," https://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf, 2008.

[14] J.-F. Raymond, "Traffic analysis: Protocols, attacks, design issues, and open problems," in *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, 2001.

[15] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg, "Effective attacks and provable defenses for website fingerprinting," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[16] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *IEEE Symposium on Security and Privacy*, 2017.

[17] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," 2010–, [Version 2.0]. [Online]. Available: https://mitmproxy.org/

[18] M. Bellare, T. Ristenpart, P. Rogaway, and T. Stegers, "Format-preserving encryption," in *Selected Areas in Cryptography*, M. J. Jacobson, V. Rijmen, and R. Safavi-Naini, Eds., 2009.

[19] D. Luchaup, K. P. Dyer, S. Jha, T. Ristenpart, and T. Shrimpton, "Libfte: A toolkit for constructing practical, format-abiding encryption schemes," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[20] Leonard Richardson, "Beautiful Soup," 2018. [Online]. Available: {https://www.crummy.com/software/BeautifulSoup/}

[21] Jakob Nielsen, "Website Response Times," https://www.nngroup.com/articles/website-response-times/, 2010.

[22] G. Sebastien, "sysstat - System performance tools for the Linux operating system," https://github.com/sysstat/sysstat, 2018.

[23] A. Janc and M. Zalewski, "Technical analysis of client identification mechanisms," https://www.chromium.org/Home/chromium-security/client-identification-mechanisms, 2018.

[24] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, M. Polychronakis, A. D. Keromytis, and E. P. Markatos, "Shadow honeypots," *International Journal of Computer and Network Security*, vol. 2, no. 9, pp. 1–15, 2010.

[25] J. Sun and K. Sun, "Desir: Decoy-enhanced seamless ip randomization," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, 2016.

[26] A. Juels and R. L. Rivest, "Honeywords: Making password-cracking detectable," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013.

[27] C. Kolbitsch, "Attacks on virtual machine emulators," https://www.lastline.com/labsblog/analyzing-environment-aware-malware/, 2014.

[28] Chrominium, "Web Page Replay," https://github.com/catapult-project/catapult/blob/master/web_page_replay_go/README.md, 2018.

[29] Telerik, "Fiddler," http://www.telerik.com/fiddler, 2018.

[30] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate record-and-replay for HTTP," in *USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[31] S. Andrica and G. Candea, "Warr: A tool for high-fidelity web application record and replay," in *IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, 2011.