SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 0000; **00**:1–12
DOI: 10.1002/sec

RESEARCH ARTICLE

Stateful Traffic Replay for Web Application Proxies

Chun-Ying Huang[1]*, Ying-Dar Lin[2], Peng-Yu Liao[2], and Yuan-Cheng Lai[3]

[1]Department of Computer Science and Engineering, National Taiwan Ocean University
[2]Department of Computer Science, National Chiao Tung University
[3]Department of Information Management, National Taiwan University of Science and Technology

ABSTRACT

It is a common practice to test a network device by replaying network traffic onto it and observe its reactions. Many replay tools support TCP/IP stateful traffic replay and hence can be used to test switches, routers, and gateway devices. However, they often fail if the device under test (DUT) is an application level proxy. In this paper, we design and implement ProxyReplay to replay application layer traffic for network proxies. As many application proxies have built-in security functions, the main purpose of this tool is to evaluate the security functionalities of DUTs using payloads constructed from real network traces. ProxyReplay modifies requests and responses and maintains queues for request-response pairs to resolve the issues of protocol dependency, functional dependency, concurrent replay, and error resistance. The solution provides two replay modes, i.e., the preprocess mode and the concurrent mode. Depending on the benchmark scenario, we show that the preprocess mode is better for benchmarking the performance capability of a DUT. In contrast, the concurrent mode is used when the replayed trace file is extremely large. Our experiments show that 99% accurately. In addition, the replay performance exceeds 320 Mbps by running the benchmark with an off-the-shelf personal computer in the preprocess mode. Copyright © 0000 John Wiley & Sons, Ltd.

KEYWORDS
Application proxy, real flow benchmarking, traffic replay

***Correspondence**
Chun-Ying Huang, Department of Computer Science and Engineering, National Taiwan Ocean University.
E-mail: chuang@ntou.edu.tw

Received . . .

1. INTRODUCTION

It is common to benchmark network devices with manually generated network traces. There are two major approaches to generating network traces manually. One is the model-based approach and the other is the trace-based approach. The model-based approach generates simulated network traces based on mathematical properties of analyzed real network traces. In contrast, the trace-based approach uses traffic replay techniques to replay previously captured network traces. There are limitations on model-based approaches. First, the correctness of simulated network traces depends on the mathematical model used to generate the traces. However, it is difficult to prove that a model is completely correct. Second, a mathematical model is often created based on well-identified network traces. Hence it is also difficult to simulate unexpected conditions, especially when an anomaly has not been revealed. Third, model-based approaches often focus only on numerical properties; it is difficult to simulate the involved application content. Finally, simulated network traces almost never act exactly the same as real network traces, especially when payloads are considered. Therefore, it is important to benchmark network devices using replay techniques so that developers and users are able to know the performance of the devices in real world.

The motivation of this work is to evaluate security functions implemented in application proxies. Therefore, one key objective is to *reconstruct application payloads precisely*. This is quite important because the implemented security functions may be triggered only when a proxy receives specific payloads. Take a web application firewall (WAF) as an example. A WAF is able to inspect common web attacks such as SQL injection, cross-site-scripting (XSS), and insecure direct object references. While such types of attacks could be embedded in any part of an HTML session, evaluating the security functions would require a complete transaction between the protected web server and its clients. Similarly, to protect end users from being compromised by attackers, an advanced

Copyright © 0000 John Wiley & Sons, Ltd.

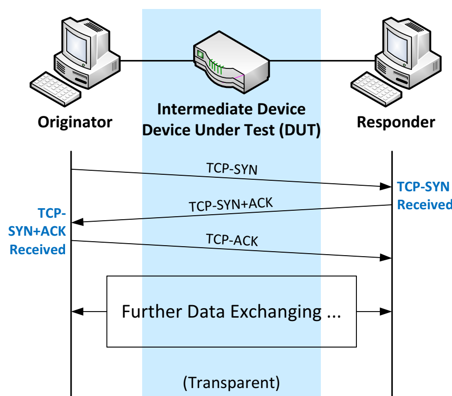


Figure 1. A regular replay scenario. The example replays a TCP flow between the traffic originator and the traffic responder.

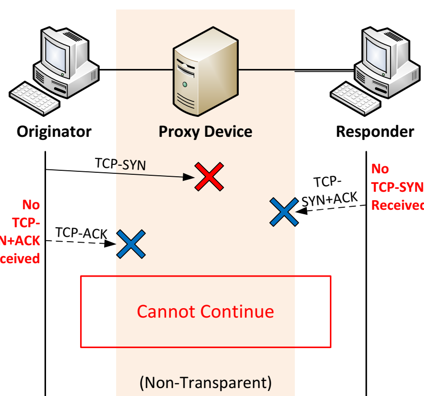


Figure 2. Replay with an intermediate proxy device. The replay fails because the connection cannot be established with the proxy. Even if the connection can be established successfully, the proxy may not understand the unmodified requests.

security gateways often implement a proxy to find out malware and viruses hidden in user downloads. In these two scenarios, evaluating the functionalities is much more critical than evaluating the performance. However, generating workloads without real exploits and real attacks do not enable us to understand the full story. Therefore, it would be better* to evaluate a security device by using replay techniques, which reproduce real exploits and attacks identified from real network traces.

Replaying network traces can be either stateless or stateful. A stateless traffic player replays network traces based only on timestamps of packets. The content of replayed network packets is exactly the same as that stored in the captured network traces. In contrast, a stateful traffic player is much more complicated. The content of replayed network packets may need to be altered to fit the network configuration. For example, the NATReplay tool [1] supports stateful traffic replay for NAT devices. Hence, it must at least maintain the mapping between private IP addresses and public IP addresses. The SocketReplay tool [2] supports stateful traffic replay for layer 4 firewall devices. It must maintain firewall responses to prevent replaying blocked connections. Compared to the above replay tools, it would be easier to replay the reconstructed payloads at the connection level instead of replaying at the packet level. To do this, payloads of TCP packets are reassembled and then delivered in TCP connections established between a replay tool and a network device. Therefore, the replay tool must be able to maintain states of each TCP connection and to resume a connection if the evaluated network device breaks the connection.

Although there are already numerous replay tools, to our knowledge, none of them are able to work with proxy devices. In general, a complete replay scenario for a proxy contains three players: the traffic originator (the client), the intermediate device (the device under test, DUT), and the traffic responder (the server). In the scenario illustrated in Figure 1, the DUT is usually transparent to the originator and the responder. Therefore, the replayed traces sent from the traffic originator are able to pass through the DUT and reach the responder, and vice versa. It is a must that the originator and the responder synchronize the replay states with each other so that the responder is able to know what should be sent back to the originator. For the ease of state synchronization, the originator and the responder are often implemented on the same machine. However, when the DUT is not transparent, an unmodified replay process could fail. Since a non-transparent proxy device only accepts inbound connections to self, the proxy rejects a connection that attempts to establish with the responder, as shown in Figure 2. Even if the proxy accepts a connection, the proxy might still not be able to handle the replayed traces because the proxy could not understand the requests. For example, a non-transparent web proxy often has to read the application protocol and the target host name from the first line of an HTTP request. However, a normal HTTP request that is sent directly to a web server does not contain these information in the request.

Most existing replay tools only maintain the states of layer 2, layer 3, or up to layer 4 [3, 4, 1, 2, 5]. Although research has shown the demands on stateful application-layer replay, these tools fail in face of application level proxies [6, 7, 8]. In order to design a replay tool that is able to accurately replay network traffic to application level proxies, four issues should be considered:

1. Protocol dependency: To replay smoothly, a connection should be established with a proxy before sending application traffic. In addition, the protocol messages may need to be modified accordingly so that the proxy is able to understand and forward the replayed content.

*Although trace-based approach has many benefits, one limitation of working with this approach is that a user has to collect sufficient amounts of diverse traces. This is also why we have set up our large scale trace collection networks, as introduced in [23].

DOI: 10.1002/sec
Prepared using secauth.cls

2. Functional dependency: Different proxies could have different behaviors, e.g., caching and content filtering [9, 10]. If a replayed content is cached or altered, the behavior must be detected and handled properly so that the replay process is not blocked.
3. Concurrent replay: Since a trace file may contain multiple concurrent connections, a replay tool must be able to replay these connections simultaneously.
4. Error resistance: Captured network traces may include a number of flaws such as packet losses, duplicated packets, and out-of-order deliveries [11]. To replay a maximum possible number of network flows, the traffic player must try to patch all identified flaws.

The proposed ProxyReplay tool is different from previous works in two manners. First, the goal of our proposed tool is to evaluate functionalities of application proxies. It must reconstruct the payloads as completely as possible, even if an application message contains errors. Although there are workload generators such as Surge [12] and Harpoon [13, 14], they are independent of web applications and hence the HTTP data payloads are filled with dummy data. Second, compare with existing replay tools, the proposed tool handles up to layer 7 protocols. Therefore, it is able to interact with an intermediate proxy device to smoothen replay processes. The proposed tool is also different from systems like [15] or [16], which provide only static content on emulated servers.

In this paper, we propose a framework to replay traffic for application level proxies. Based on the framework, an HTTP traffic replay tool named ProxyReplay is implemented to show its effectiveness and efficiency. The rest of this paper is organized as follows. Related work regarding traffic capture and traffic replay is introduced in Section 2. The design and the implementation of the proposed solution are introduced in Section 3 and Section 4, respectively. Evaluation of the proposed solution is performed in Section 5. Finally, a concluding remark is given in Section 6.

2. RELATED WORK

2.1. Type of Application Proxies

We can classify application level proxies into two types, i.e., non-transparent and transparent proxies. The major difference between the two types of proxies is the client awareness of the intermediate proxy. Figure 3 shows the application scenarios for the two types of proxies. In the two scenarios, there are three roles, i.e., the client A, the proxy P, and the server B. With a non-transparent proxy, the client A has to know the address of the proxy P. When A is going to interact with a remote server B, it sends a request to P instead of B. The proxy P then forwards the request received from A to B. After server B has processed the request, B sends a response back to the proxy P and then the proxy forwards the response to A as well. In this scenario, as shown in Figure 3(a), the communication channel between A and B is split into two independent connections. One is established between A and P and the other is established between P and B.

In contrast, with a transparent proxy, the client A does not need to know the address of the proxy P. As shown in Figure 3(b), although the communication channel between A and B is also split into two independent connections, A does not know that the intermediate proxy P intercepts its request. The connection established between P and B is exactly the same as that in the non-transparent scenario. However, before P forwards the response from B to A, it has to masquerade as B.

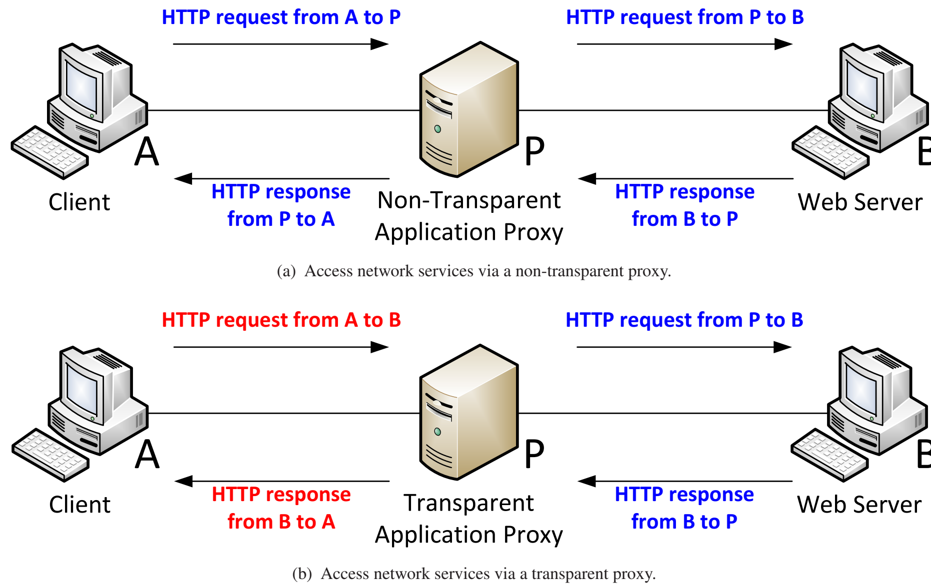
Modern application level proxies such as Squid [17] can be configured as either a non-transparent proxy or a transparent proxy. However, existing replay tools may not work with application level proxies even if a proxy is configured as a transparent one. For the ease of comparison, both the functional and the performance evaluations of each type of proxy will be made with the open source Squid proxy software.

2.2. Existing Replay Tools

"Capture and then replay" is one efficient strategy for testing and evaluating network applications [18]. There are numerous tools developed for traffic replay. The TCPReplay [3] tool replays packets based only on timestamps. It does not interact with the intermediate device. In contrast, the Tomahawk [4] tool replays packets in a sequential manner. It replays the next packet right after the response for an earlier replayed packet is received. The above two tools are representatives of stateless traffic replay tools. To speed up packet-level replay, researchers also attempt to develop hardware-based replayers such as ITester [19].

There are also several stateful traffic replay tools. Stateful replay tools may handle states at different network layers. TCPopera [5] is a stateful replay tool that emulates the TCP/IP stack. Hence, it ensures zero "ghost packet" generation, which is a critical feature for the test environments where the accuracy of protocol semantics is of fundamental importance. The SocketReplay [2] tool retrieves partial TCP payloads from captured network flows and replays through self-established connections. The SocketReplay tool has to store all identified TCP flows in its own format. To save storage spaces, the authors did not store the entire payloads. Instead, only the first several bytes of a flow are preserved. The NATReplay [1] tool maintains the mapping state between private IP addresses and public IP addresses so that traffic can be replayed smoothly through NAT devices. The above three tools maintains only layer 3 and layer 4 network states. They are not able to work with proxy devices.

There are replay tools that handle layer 3, layer 4, and layer 7 network states. RolePlayer [7] identifies specific



(a) Access network services via a non-transparent proxy.

(b) Access network services via a transparent proxy.

Figure 3. Application scenarios for non-transparent and transparent proxies.

network protocol attributes such as IP addresses and host names using machine-learning algorithms. It then modifies these identified attributes accordingly to replay application protocols. Replayer [8] uses binary analysis and program verification techniques to solve the problem of replaying at the application layer. The system is able to guarantee accurate responses to requests received based on the result of protocol state analysis. Monkey [6] focuses on testing web servers. Two programs, *Monkey See* and *Monkey Do* are used to capture and replay HTTP network traffic, respectively. TCP connections are established by emulated web clients and web servers to perform stateful traffic replay.

Although the above replay tools handle layer 7 states, they cannot replay smoothly in the presence of proxy devices. As we have discussed in Section 1, a tool that is designed for transparent network devices cannot be used with proxies. Therefore, in this paper, we design and implement *ProxyReplay*, a tool that aims to replay captured access traffic through proxies. A brief comparison of existing replay tools and the proposed *ProxyReplay* tool is shown in Table I. Note that ProxyReplay is not designed to replace other tools. It is used to complement the demand for benchmarking proxy devices.

Table I. Comparison of existing replay tools and the proposed solution.

Name	Replay broken trace	Is stateful?	Supported devices
TCPReplay	Yes	No	non-proxy
NATReplay	No	Layer 4	non-proxy
SocketReplay	Yes	Layer 4	non-proxy
Monkey	Yes	Layer 7	non-proxy
RolePlayer	No	Layer 7	non-proxy
Replayer	No	Layer 7	non-proxy
ProxyReplay	Yes	Layer 7	proxy

3. THE DESIGN OF THE PROPOSED SOLUTION

3.1. Architecture with Queues for Request-Response Pairs

Figure 4 shows the system architecture of ProxyReplay. There are two major components, namely the parser and the replayer. The parser is used to parse raw packet traces into internal formats and the replayer replays the traces based on the parsed information. At the beginning, the parser loads a pcap [20] trace file as the input and converts the trace into internal data structure. Based on a five-tuple†, packet payloads are assembled into streams first. Each stream is segmented into several request-response pairs. The request-response pairs are labeled and appended into the client queue and the server queue of the replayer. In addition to generate request-response pairs, all domain names involved in the requests are stored in a domain name service database for future use.

To replay the parsed data, the replayer iteratively picks up a request from the client queue, establishes connections with the DUT, and replays the request to the DUT. When the request is sent out from the originator, passes through

†The five-tuple consists of source IP address, destination IP address, source TCP/UDP port number, destination TCP/UDP port number, and the transport layer protocol.

Security Comm. Networks 0000; **00**:1–12 © 0000 John Wiley & Sons, Ltd.
DOI: 10.1002/sec
Prepared using secauth.cls

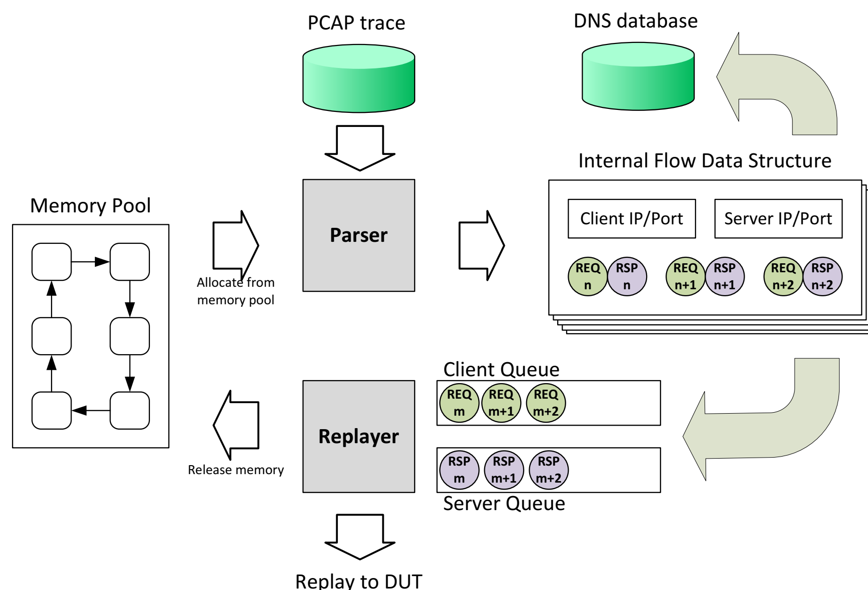


Figure 4. The system architecture of ProxyReplay.

the DUT, and reaches the responder, a corresponding response is then replayed back to the DUT. The parser and the replayer work concurrently until packets in the pcap trace file are all processed and all the identified request-response pairs are replayed.

Readers should note that, to be memory efficient, the replay tool allocates a fixed-size memory pool to store identified request-response pairs. When a new request-response pair is identified, required memory spaces are allocated from the memory pool. Once a request-response pair is replayed, the memory spaces used by that pair should be returned to the pool. The proposed solution has two different replay modes (introduced later in Section 3.2.3). In the concurrent mode, required memory spaces depend on the request arrival rate, request departure rate, process time, and request sizes. These parameters are different from trace to trace. In contrast, when flows are replayed in the preprocess mode, all the parsed data must be able to be stored in the memory pool. Therefore, a trace file that is greater than the size of the memory pool cannot be replayed.

3.2. Solutions to Application Level Proxy Replay Issues

3.2.1. Protocol Dependency: Request/Response Modification with DNS Emulation

We solve the protocol dependency issue by replaying requests and responses in a stateful manner. When a request-response pair has been identified from a trace, the destination IP address and the port number are modified to meet the configuration of the proxy-under-test first. Then, parts of the request may need to be altered so that the request can be recognized and forwarded by the proxy. For example, suppose a client sends a request to a web server `www.sample.com` on port 80 and retrieves the welcome page using the `GET / HTTP/1.1` request. The request must be changed to `GET http://www.sample.com/ HTTP/1.1` so that a non-transparent proxy is able to understand the request. Finally, when a proxy decided to forward the request, it may make a domain name lookup by contacting a DNS server. For example, looking up the IP address of the domain name `www.sample.com` in the above example. Hence, we need to emulate a DNS server as well to answer requests sent by proxies. The responded IP address is always set to the server-side address of the replayer. A complete scenario of a stateful replay is given in Figure 5. Note that the emulated hosts A, B, and D are parts of ProxyReplay and they are all implemented on the same machine. The host P is the device (proxy) under test. Therefore, there are only two machines in the benchmark scenario.

Although there are tools, such as Bro [21], that are able to extract request-response pairs, we still implement our own parser for the following three reasons. First, most existing tools have to store parsed results in files. This would slow down the replay performance and increase the costs of storage spaces. To be more efficient, the proposed solution stores parsed intermediate data only in memory. Second, even if third-party tools are able to decode the HTTP request, responses, and payloads, we still have to parse the decoded results again to retrieve the data used to build DNS databases. With our own parser, we can finish both parsing and DNS database construction in one shot. Finally, some existing tools parse application protocols using scripts. Although they provide great flexibilities, they

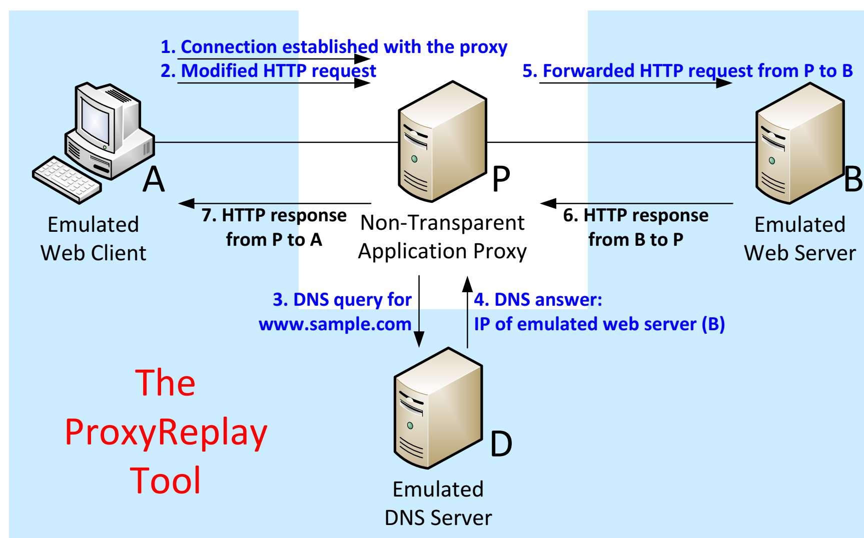


Figure 5. A scenario of stateful proxy replay. Readers should note that all components are implemented in a single machine with multiple network interfaces.

may not run as efficiently as a parser written in native C/C++ codes.

3.2.2. Functional Dependency: On The Fly Content Modification

A proxy under test could apply any arbitrary policies to the replayed traffic; thus, to guarantee that the traces can be smoothly replayed to the proxy, the replay tool may have to interact with the proxy or modify the replayed content on-the-fly. Here we provide two scenarios to illustrate the problems and the solutions.

(1) *Replay to a filter proxy.* A filter proxy is often used to check whether a request or a response is allowed or not. If a request or a response is disallowed, it is blocked and an error message is sent to the requesting client. The problem of such behavior is that, when a replayed request sent by the originator (client side) *does not reach* the responder (server side), the memory occupied by the request-response pair will never be released. Although it is possible to release the memory spaces by using an expiry timer, available memory spaces could be depleted quickly when a large number of requests are blocked in a short period of time.

To avoid this problem, the replay tool has to read responses from the proxy and verify whether the responder or the proxy itself returns the response. When the proxy itself returns a response, it is most likely that the request is rejected or blocked by the proxy. On receipt of the rejection, the replayed request-response pair should be removed from the queue as well.

(2) *Replay to a cache proxy.* A cache proxy is often used to save the outbound bandwidth consumed by identical requests. However, similar to the problem of replaying with a filter proxy, memory spaces could be depleted if requests cannot reach the responder due to cache hits. In addition to the memory management problem, replaying with a cache proxy has another issue. Suppose that a user plans to benchmark the performance of a cache proxy when cache always misses. The replay tool must guarantee that the proxy caches no responses even if a trace contains identical request-response pairs. In the HTTP protocol, the replay tool can prevent responses from being cached by setting the “Expires” header to a past time or using the “Cache-Control” header to disable the cache function.

It is not possible to enumerate all policies implemented by a proxy. To maximize the extensibility of the proposed solution, the implementation should allow users to develop their own plug-ins to verify and modify requests/responses.

3.2.3. Concurrent Replay: Replay at a Relative or Full Speed

The issue of concurrent replay is an implementation issue. To support diverse benchmarking scenarios, the proposed solution provides two modes to schedule how identified requests are replayed. First, an identified request can be replayed at a relative time to the first identified request. Thanks to the multiplex design of network socket implementation, by using system calls like `select` and `poll`, one replayer should be enough to replay all requests at the scheduled replaying time. We call this mode the *concurrent mode*. The replay performance of the concurrent mode depends on how powerful the CPU and the network interface is. If a single replayer cannot meet performance requirements, we can run more replayers

Security Comm. Networks 0000; **00**:1–12 © 0000 John Wiley & Sons, Ltd.
DOI: 10.1002/sec
Prepared using secauth.cls

concurrently on different CPUs and let all replayers share the same client queue and server queue. Note that the proxy replay tool *does not control the timing of the replayed requests, but it guarantees that the order of the replayed requests is exactly the same as the captured requests*. This is because the purpose of this tool is to *evaluate the functionalities* of security devices. The timing usually does not affect the detection of an anomaly event but the order of requests would significantly affect the detector. Note that the timing granularity of the concurrent mode is in seconds. However, the occurrence order of requests is kept the same if the relative time is less than one second. Readers should also note that the timing is controlled in a per-connection basis instead of a per-packet basis because payloads of packets of the same TCP connection are reassembled as a stream before they are replayed. Hence, a reassembled stream starts to be replayed at the relative time of seeing the first packet of a connection. For example, suppose a request is captured at time T and replayed at time t, a subsequent request captured at time $T+\Delta t$ would be scheduled to replay at time $t+\Delta t$. However, if the parser were not able to finish the parsing before time $t+\Delta t$, the parsed request would be replayed immediately.

Alternatively, a user may plan to replay all identified data streams at full speed to benchmark the performance of a proxy. With this mode, requests are replayed back-to-back. To make sure that requests can be replayed back-to-back, all requests must be identified and then stored in the memory so that they can be replayed without any delay. We call this mode the *preprocess mode*. The replay performance of the preprocess mode depends more on the I/O performance of the replay machine. In addition, since all identified requests have to be stored in the memory, there must be enough memory spaces to store all the data.

One technique to improve the replay performance for both the concurrent mode and the preprocess mode is to convert a pcap trace file into a preprocessed intermediate file format. Then, the replayer is able to replay the intermediate file directly without parsing the pcap file again. This would be a good trade-off between storage costs and computation costs. However, we do not store intermediate results because we have an extremely large dataset of network traces stored in pcap format. Storing intermediate formats would double the storage spaces and hence the proposed solution only supports on-the-fly replay mode.

3.2.4. Error resistance: Handling Packet Retransmission, Out-of-Order Delivery, and Packet Loss

As the system architecture shows, replayed requests and responses are parsed and extracted from packet traces. However, several common network issues could affect the correctness of reconstructing network flow. These issues include packet retransmissions, out-of-order deliveries, and packet losses. These issues must be handled properly so that the request-response pairs can be parsed and extracted correctly.

It is easier to handle packet retransmissions and out-of-order deliveries than packet losses. By inspecting the sequence numbers, if there are duplicated packets, they can be simply ignored. If packets are delivered out-of-order, they can be reordered based on sequence numbers as well. Packet retransmission and out-of-order delivery can be easily handled by leveraging open source codes like `libnids` [22]. However, if there is an unrecoverable packet loss in a TCP connection, we simply ignore all the packets belonging to the connection. Although it is possible to estimate the length of payloads by inspecting the TCP sequence numbers or reading from the application protocol's content length header, it is not possible to reconstruct the payloads in the lost segment. In this case, we choose not to reassemble the flawed flow.

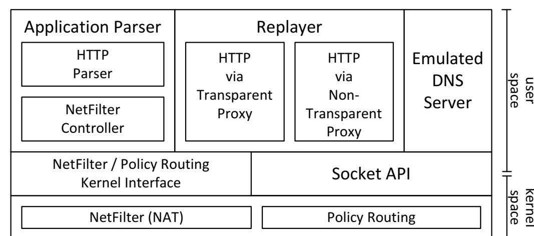


Figure 6. The software components of the ProxyReplay tool.

4. IMPLEMENTATION

Figure 6 shows the software components of ProxyReplay. ProxyReplay is implemented on the Linux operating system. The components are implemented in the user space with an additional controller to configure kernel configurations on demand. The components include the parser, the replayer, and the emulated DNS server. To allow a single machine emulating both clients (originators) and servers (responders) simultaneously, we have to make sure that communications between the server and the client are sent on the wire instead of sending via the loop back path. ProxyReplay controls the policy routing component and the netfilter NAT module to achieve the goal. The issues related to these components are discussed in the following subsections.

4.1. The Parser

The parser is implemented in $1+n$ threads. One thread is dedicated to read packets from the input pcap trace file. The other n threads are used to reassemble packet payloads. To guarantee that packets belonging to the same connection are processed by the same worker thread, we use client source port number as the key to dispatch jobs. The parsed request-response pairs are stored in the memory pool. If all memory spaces in the memory pool are

depleted, the parser pauses and waits until more memory spaces become available. The parser stops when an entire pcap file is processed.

In addition to reassemble packet payloads into data streams, the parser has to update the DNS database and configure the kernel properly. The domain name of a web server can be retrieved from the HTTP request header and the server address can be obtained from the IP packet header. If the domain name were available, a corresponding DNS entry would be created in the database of the emulated DNS server. The IP address of the server is updated into the kernel accordingly to make sure the replay process would not be blocked. Please refer to Section 4.3 for the details of on-demand kernel configurations.

4.2. The Replayer

There are two issues related to the replayer. First, a single replayer must have the ability to manage a number of concurrently replayed connections. In the Linux operating system, each network connection is represented by a file descriptor. To have a better performance, we use the `epoll` system call to handle multiple descriptors within a single thread. Compare with the `select` and the traditional `poll` system call, `epoll` has no limit on the number of file descriptors. In addition, the performance of `epoll` is independent of the number of file descriptors.

Another issue for the replayer is the timing of replay. The replayer is able to replay request-response pairs immediately when a pair is available in the client queue and the server queue. Alternatively, the replayer can choose to wait until an entire pcap file is processed or the available memory spaces for the client queue and the server queue are depleted. The benefit of the former choice is that there is no size limitation on the replayed trace file. However, the replay performance is bounded by the performance of the parser, i.e., the number of requests that can be parsed per second. In contrast, the benefit of the latter choice is that the replayer is able to replay at its maximum performance. Nevertheless, the size of the trace file is limited by the total available memory spaces. If a benchmark scenario only plans to validate functionalities of a proxy, choosing the former one would be fine. However, if a benchmark scenario were intended to understand the best throughput of a proxy, the latter one would be a better choice. ProxyReplay implements both the above two choices. In the implementation, we call the former one *the concurrent mode* and the latter one *the preprocess mode*.

4.3. Kernel Configurations

Since ProxyReplay emulates clients and servers on the same machine, the operating system kernel has to be configured properly to make sure that the replay can be done on the wire, not through the loopback path. The host machine running ProxyReplay must install two network interface cards. One is used to emulate the client side and the other is used to emulate the server side. To prevent these addresses from conflicting with Internet addresses, the two network interfaces are configured with private IP addresses residing in two different subnetworks. For the ease of discussion, we name the two IP addresses IP_c and IP_s for client side interface and server side interface, respectively.

Readers should note that when a reassembled request-response pair is replayed, the client side does not use the client IP address of the request; instead, the server side is emulated with the server address of the request. Suppose a parsed request-response pair has a client address of RIP_c and a server address of RIP_s. The replayed TCP connection would have a client address of IP_c and a server address of RIP_s. The domain name and the IP address of the server should be preserved during the replay process because many security mechanisms identify anomalies by using black and white lists, which are built upon domain names and IP addresses of remote servers.

To preserve server IP addresses, one naïve solution is to use IP aliases on the replay machine. However, this is not practical because adding and deleting IP aliasing on demand is relatively cost-ineffective. To be more efficiently, we use policy routes and NAT techniques to achieve the same effect. The policy routing is used to guarantee that all requests sent from the client side are placed on the wire. In a non-transparent replay scenario, this can be done by simply routing all unknown targets to the default gateway or the proxy. In a transparent replay scenario, it can be done by creating pseudo-gateways using static ARP binding. For the NAT part, in both non-transparent and transparent replay scenarios, each retrieved server address must have a corresponding destination NAT rule to map the server address, RIP_s, to the address of the server side interface, IP_s. Compared to IP aliasing, operating NAT rules are much more cost-effective.

A complete configuration for replaying in a transparent scenario is given in Figure 7. We use equivalent Linux command line tools to illustrate the configuration. The notations used in the scenario are listed below:

- `eth1` and `eth2`: The interface names of the client side and the server side interface.
- IP_c and IP_s: The IP addresses of the client side and the server side interfaces, respectively.
- MAC_c and MAC_s: The hardware addresses of the client side and the server side interfaces, respectively.
- GW_c and GW_s: The pseudo-gateway addresses of the client side and the server side interfaces, respectively.
- RIP_s: The server address of a replayed request.

Note that only Line 7 of the configuration is required for each identified server IP address during a replay process. The other configurations can be done only once at the very beginning of the benchmark. If a server address is no longer referenced, it can be removed from the NAT rule.

Security Comm. Networks 0000; **00**:1–12 © 0000 John Wiley & Sons, Ltd.
DOI: 10.1002/sec
Prepared using secauth.cls

```
1: ip route add $GW_c dev eth2
2: arp -i eth2 -s $GW_c $MAC_c
3: ip route add $GW_s dev eth1
4: arp -i eth1 -s $GW_s $MAC_s
5: ip route change default via $GW_s
6: iptables -t nat -A POSTROUTING -s $IP_c -j
SNAT --to-source $GW_c
7: iptables -t nat -A PREROUTING -d $RIP_s -j
DNAT --to-destination $IP_s
```

Figure 7. The configuration used to replay a parsed request-response pair for a transparent proxy.

Table II. The hardware and software specifications of machines used in the evaluation environment.

	Proxy Under Test	Replay Machine
CPU	AMD Athlon 64 3000+	Intel Core 2 Quad Q8200 2.33GHz
RAM	1 gigabytes	4 gigabytes
OS	Linux x86_64†	Linux x86_64†
Network Interfaces	Two interfaces in bridged mode	eth0: emulated client eth1: emulated server
Software	Squid 3 series	The ProxyReplay tool

†: The Linux kernel version we work with is 2.6.35.

Table III. Statistics of the major pcap trace file.

Item	Statistics
Size	2.6 gigabytes
Elapsed time	20 minutes
# of packets	3,168,672
# of connections	10,964
# of clients	363
# of servers	916
# of request-response pairs	21166
# of requests for `jpg` files	7,134
# of requests for non-`jpg` files	14,032

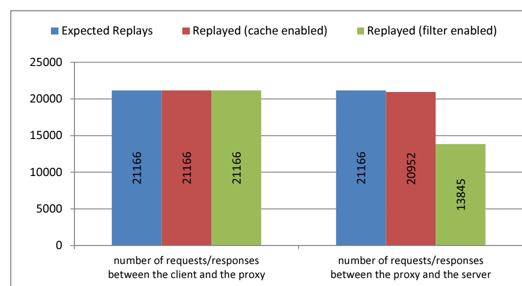


Figure 9. Results of replaying to a cache proxy and a filter proxy.

5. EVALUATION AND DISCUSSION

We evaluate the functionality, performance, and scalability of ProxyReplay using HTTP traces. The evaluation environment is introduced first in Section 5.1. The scenarios and results for evaluating its functionality, performance, and scalability are given in Sections 5.2, 5.3, and 5.4, respectively. Each experiment conducted in this section is repeated for 10 times and the results are the average performance of the proposed solution. A comparison between the solution and other well-known replay tools is given in Section 5.5. Finally, the availability and the extensibility of ProxyReplay is discussed in Section 5.6.

5.1. Evaluation Environment

Figure 8 and Table II show the environment, the software specifications, and the hardware specifications of host machines used to evaluate the implemented ProxyReplay tool. There are three roles: the HTTP proxy server, the replay machine, and the network sniffer. The HTTP proxy server runs the open source *squid* software. The replay machine runs the ProxyReplay tool. All machines run the Linux operating system.

In addition to the hardware and software configuration, we need an HTTP trace file to perform all the evaluations. The trace is collected from BetaSite network [23]. It is a 20-minute HTTP-only packet trace and the size is 2.6 gigabytes. A summary of statistics about the trace is given in Table III. We choose a relatively small pcap file because the file can be completely stored in memory. This would be easier for us to compare the performance of the concurrent mode and the preprocess mode.

5.2. Functionality

To understand whether the proposed ProxyReplay tool works well with proxies with different functionalities, we configure the squid proxy server as a cache proxy and a filter proxy. Then, we use a network sniffer to capture the replayed traffic and compare the captured trace against the original trace to verify the correctness of the replay process.

We assume that the proxy under test is a black box. When testing with a cache proxy, we use the default configuration provided in the squid software. Similarly, when testing with a filter proxy, in addition to the default configuration, we simply add a customized rule to filter requests to `jpeg` files. Figure 9 shows the results of replaying to the proxies. Independent of the use of a proxy, the number of messages exchanged between the client and the proxy is always identical to the number of all parsed request-response pairs. When the cache feature is enabled, we can see that the proxy forwards all requests, except the 1% Similarly, when the traces are replayed through a filter proxy, we can also observe that the proxy forwards only non-`jpg` requests. Note that the number of forwarded non-`jpg` requests is slightly lower than the number of detected non-`jpg` requests because the intermediate proxy caches those missing requests.

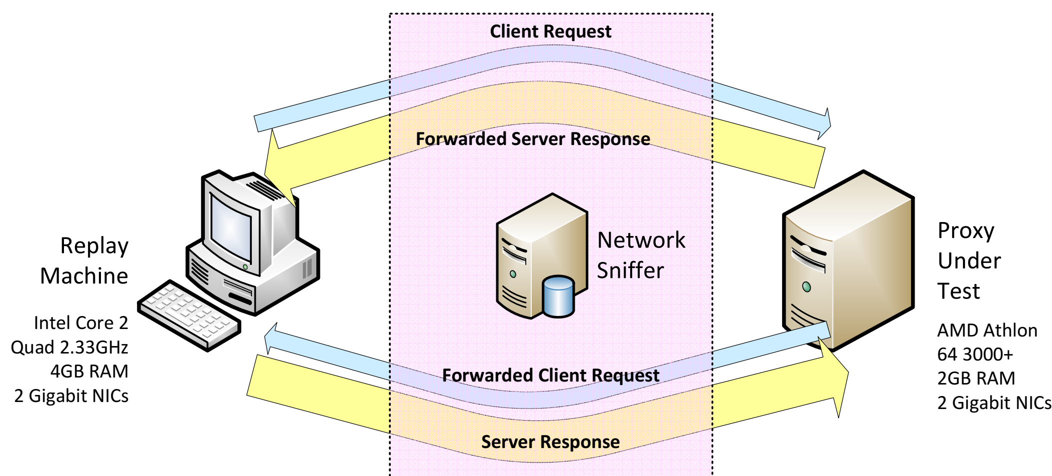


Figure 8. The evaluation environment for ProxyReplay.

Table IV. Replay performance of the two different replay modes.

	preprocess mode	concurrent mode
Parse time	Depends on size of traces	2 seconds (initial setup)
Average throughput	320 Mbps	120 Mbps*
File size limits	By memory spaces	Not limited
Test #1 (800 MB)	passed	passed
Test #2 (2.6 GB)	passed	passed
Test #3 (5.2 GB)	failed	passed
Test #4 (2.2 TB)	failed	passed
Benefits	Good performance	Short parse time No size limits
Drawbacks	Long parse time Size limits	Low performance

$*$: The average throughput of the input trace is 150 Mbps.

5.3. Performance

We evaluate both the performance of the preprocess mode and the concurrent mode in terms of the replayed throughput and the replayed request rate. Table IV shows the statistics for the performance benchmarks. We can see that the best replay throughput in the given environment is 320 Mbps using the preprocess mode. Note that the parser performance in the concurrent mode dominates the replayed throughput and the request rate. The request rate depends on the parsing rate, i.e., the number of request-response pairs that can be parsed by the parser in one unit of time. However, the request rate is non-deterministic because a parser cannot determine when a response terminates. A response can be transmitted within one packet and it is also possible to be transmitted using a number of packets. When a response lasts longer, the parsing time becomes longer and hence the parsing rate decreases.

The replayed throughput depends on the payload length of the replayed requests and responses. It is also non-deterministic since payload lengths are not predictable. Suppose a replayed connection is used to replay only a single request-response pair. If the total payload length of the request and the response were shorter, the throughput would decrease because most of the time is spent on establishing new connections. If the total payload length were longer, the throughput would be higher.

5.4. Scalability

We further evaluate the scalability of the proposed ProxyReplay tool. In addition to the major pcap trace file, we use another two different sizes of pcap trace files to benchmark the scalability of the proposed tool. Table IV also shows a comparison of replayed results using the two different replay modes. We show that even a sizeable (2.2TB) dataset can be replayed smoothly in the concurrent mode. Note that the installed physical memory on the replayer is 4 gigabytes. If a trace file size exceeds that

Table V. Throughput comparison of the proposed solution against other publicly available replay tools.

	TCP Replay	NAT Replay	Socket Replay	Proxy Replay
Direct	363 Mbps	–	40 Mbps	–
Router	288 Mbps	–	12 Mbps	–
NAT	–	320 Mbps	–	–
Proxy	–	–	–	320 Mbps

value, it cannot be replayed. Readers should note that although it is possible to install more than 4 gigabytes of physical memory on a 32-bit machine, the address space available to a user space process is still limited to 4 gigabytes. Therefore, to replay a really large trace file using the preprocess mode, it would be better using a 64-bit machine and a 64-bit operating system. Otherwise, we suggest using the concurrent mode to replay large trace files.

5.5. Comparison with Other Replay Tools

We also compare the proposed ProxyReplay tool with other well-known replay tools. We select the tools based on two criteria. First, the tool must be able to handle at least layer 4 protocols; and second, the tool must be able to replay captured payloads instead of emulated or dummy payloads. All the tools are benchmarked using the same hardware environment and the same trace. However, since a replay tool may only work for some specific network configurations, the configurations are slightly different between each test. Since the targeted applications of each replay tool are diverse, we focus only on the replayed performance. The comparison of the performance between ProxyReplay and other tools can be found in Table V. It is straightforward that TCPReplay and NATReplay have the best performance. This is because they maintain a minimal set of states and the modifications can be done at the packet level. SocketReplay and ProxyReplay are both replayed by establishing real network connections. The performance gap between SocketReplay and ProxyReplay could be due to implementation problems.

5.6. Availability and Extensibility

We have released the source codes of ProxyReplay and relevant our developed benchmark tools to the public since 2012. The codes are hosted on OpenFoundry, the repository of our domestic open-source promotion project. Interested readers are able to access the source codes of ProxyReplay from *http://www.openfoundry.org/of/projects/2026/download*. Although ProxyReplay is currently designed for web application proxies, it can be easily extended for other types of application proxies. In short, the handling of packet I/O, connection setup, memory management, and DNS emulation can be reused and a developer or a researcher can focus more on the parsing and mangling of layer 7 content if necessary.

6. CONCLUSION

We design and implement ProxyReplay to statefully replay network traces to application level proxies. The proposed tool provides two replay modes, *the preprocess mode* and *the concurrent mode*, that can be used to benchmark the functionality and the performance of a proxy device, respectively. The proof-of-concept implementation of ProxyReplay shows that it works well with real world traces and proxies. Although we only discuss the case of replaying HTTP traffic, the proposed architecture can be applied to other application protocols. We believe that ProxyReplay is able to complement the demanding requirements for benchmarking complicated application level proxy devices.

ACKNOWLEDGMENT

This research was supported in part by National Science Council in Taiwan, and in part by ZyXEL Corp. and D-Link Corp. We would also like to thank the anonymous reviewers for their valuable and helpful comments.

REFERENCES

1. Network Benchmarking Lab. NATreplay test tool 2009. URL `http://www.nbl.org.tw/`, [online] `http://www.nbl.org.tw/`.
2. Lin YD, Lin PC, Cheng TH, Chen IW, Lai YC. Low-storage capture and loss recovery selective replay of real flows. *IEEE Communications Magazine* April 2012; **50**(4):114–121.
3. Turner A. TCP Replay: pcap editing & replay tools for UNIX April 2010. URL `http://tcpreplay.synfin.net/`, [online] `http://tcpreplay.synfin.net/`.
4. Smith B. Tomahawk test tool 2006. URL `http://tomahawk.sourceforge.net/`, [onlne] `http://tomahawk.sourceforge.net/`.
5. Hong SS, Wu SF. On interactive internet traffic replay. *RAID 2005: Proceedings of Recent Advances in Intrusion Detection*, 2005.
6. Cheng YC, Hlzle U, Cardwell N, Savage S, Voelker GM. Monkey see, monkey do: A tool for tcp tracing and replaying. *USENIX04: Proceedings of USENIX 2004 Annual Technical Conference*, 2004; 87–98.
7. Cui W, Paxson V, Weaver NC, Katz RH. Protocol-independent adaptive replay of application dialog. *NDSS06: Proceedings of the 13th Annual Network and Distributed System Security Symposium*, 2006.

8. Newsome J, Brumley D, Franklin J, Song D. Replayer: automatic protocol replay by binary analysis. *CCS06: Proceedings of the 13th ACM conference on Computer and communications security*, ACM, 2006; 311–321.
9. Pierre G, Makpangou M. Saperlipopette!: a distributed web caching systems evaluation tool. *Middleware'98: Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, 1998; 389–406.
10. Wessels D. *Web Caching*. O'Reilly Media, 2001.
11. Schneider F, Wallerich J, Feldmann A. Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware. *PAM07: Proceedings of Passive and Active Network Measurement*, 2007; 207–217.
12. Barford P, Crovella M. Generating representative web workloads for network and server performance evaluation. *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ACM, 1998; 151–160.
13. Sommers J, Barford P. Self-configuring network traffic generation. *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, ACM, 2004; 68–81.
14. Sommers J, Kim H, Barford P. Harpoon: a flow-level traffic generator for router and network tests. *SIGMETRICS Performance Evaluation Review* June 2004; **32**(1):392–392.
15. Lin YD, Shih TB, Wu YS, Lai YC. Secure and transparent network traffic replay, redirect, relay in a dynamic malware analysis environment. *Security and Communication Networks* March 2013; .
16. Bscher A. Forensic tool to replay web-based attacks (and also general http traffic) that were captured in a pcap file. `https://code.google.com/p/replayproxy/`.
17. Squid. squid: Optimising web delivery. URL `http://www.squid-cache.org/`, `http://www.squid-cache.org/`.
18. Leotta M, Clerissi D, Ricca F, Tonella P. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*, IEEE, 2013.
19. Zhang F, Xie Y, Liu J, Luo L, Ning Q, Wu X. ITester: A FPGA based high performance traffic replay tool. *Proceedings of the 22nd IEEE International conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2012; 699–702.
20. Lawrence Berkeley National Laboratory. tcpdump/libpcap. URL `http://www.tcpdump.org/`.
21. Paxson V. Bro: a system for detecting network intruders in real-time. *Computer Networks* December 1999; **31**(23–24):2435–2463.
22. Wojtczuk R. libnids. URL `http://www.squid-cache.org/`, `http://libnids.sourceforge.net/`.
23. Lin YD, Chen IW, Lin PC, Chen CS, Hsu CH. On campus beta site: Architecture designs, operational experience, and top product defects. *IEEE Communications Magazine* December 2010; **48**(12):83–91.

Security Comm. Networks 0000; **00**:1–12 © 0000 John Wiley & Sons, Ltd.
DOI: 10.1002/sec
Prepared using secauth.cls