Measuring and Analyzing DoS Flooding Experiments

Amir Farhat
Massachusetts Institute of Technology
Cambridge, MA, United States
amirf@mit.edu

Samuel DeLaughter
Massachusetts Institute of Technology
Cambridge, MA, United States
samd@mit.edu

Karen R. Sollins
Massachusetts Institute of Technology
Cambridge, MA, United States
sollins@csail.mit.edu

ABSTRACT

In running volumetric Denial-of-Service experiments on DeterLab, we have faced challenges regarding the collection and handling of experiment results without impinging on the performance of the experiments themselves. In this paper we report on our findings with respect to both the collection and handling of data from our experiments, based on challenges of both performance and scale. To instrument experiments with minimal interference with results, we use simple packet capture, in contrast with DeterLab's supported real-time collection and insertion of experiment data, which is prohibitively slow for our high-volume and many-experiment use case. The contribution of this work is a comparison of two approaches to experiment grouping and analysis driven by the example of our experimental requirements of high-volume and performance sensitivity.

CCS CONCEPTS

• **General and reference → Measurement**; **Performance**; Experimentation; • **Information systems → Relational database model**; *Database performance evaluation*; • **Security and privacy → Denial-of-service attacks**.

KEYWORDS

Denial of Service, Experiment Instrumentation, Storage Optimization, Query Optimization

ACM Reference Format:
Amir Farhat, Samuel DeLaughter, and Karen R. Sollins. 2022. Measuring and Analyzing DoS Flooding Experiments. In *Cyber Security Experimentation and Test Workshop (CSET 2022), August 8, 2022, Virtual, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3546096.3546105

1 INTRODUCTION

In this paper we present a novel framework for collecting and analyzing data from high-volume Denial-of-Service (DoS) flooding attacks in security experimentation testbeds. The design and implementation of our framework is motivated by our heavy use of DeterLab for DoS-resilience research in the IoT space, but we have observed that this framework is a good fit for other high-volume networked experiments [4]. DeterLab is effective for launching security experiments on physical machines, but its data measurement, storage, and analysis is ill-fitting for high-volume and many-experiment use cases. Our experiments produce 1GB of network data on average (10GB in the largest case) and we run roughly 100 experiments. We group experiments together and analyze them at the group level to draw conclusions about the DoS-resilience of different configuration parameters of our system, as shown in Table 2. Most of our experiment sizes come from Gbps attack rates potentially lasting minutes. We are able to efficiently process our many high-volume experiments by employing low-overhead instrumentation, leveraging parallelism, and choosing good storage methods for experiment results. In particular, we propose two ways to store experiment groups: file-based and database-assisted, preferring the latter for more scalable experiment groupings.

In what follows, we provide background on DeterLab and our security experiments in Section 2, discuss the design choices, tradeoffs, and implementation considerations of our framework in Sections 3 and 4, evaluate the performance and usability of the framework in Section 5, discuss future work and applicability of this framework to other experiments in Section 6, and finally conclude in Section 7.

CSET 2022, August 8, 2022, Virtual, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9684-4/22/08.
https://doi.org/10.1145/3546096.3546105

2 BACKGROUND

This section provides background information on DeterLab and the nature of the experiments we run. We highlight why these experiments pose a challenge for measurement and storage, and how they are applicable to other cybersecurity experiments.

2.1 DeterLab

DeterLab allows researchers and educators to allocate real machines connected with real, physical networks for the purpose of running cybersecurity experiments without leaking any malicious traffic out to the Internet [4]. Real machines are important for accuracy of experiment results to mirror the real deployments they simulate. Leaking malicious traffic is dangerous because it could victimize organizations and is illegal [18].

DeterLab offers tools for running experiments, such as the MAGI orchestrator, which provides "a workflow management system for DeterLab ... and control and data management for experiments" [10]. The MAGI orchestrator includes a data management layer to store experiment data in MongoDB, a scalable collection-oriented NoSQL database [8, 9, 17]. The MongoDB instance is allocated on hardware separate from experimentation machines and is the only option for data storage. MAGI's approach to experimentation couples data measurement with database insertion at experiment run-time [9]. The large data volume in our experiments, along with the instrumentation overhead, make it infeasible to insert data into the database in real time.

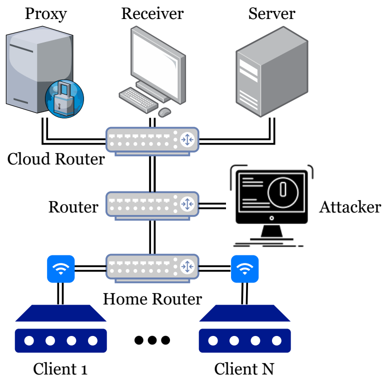

Figure 1: Diagram depicting a standard topology of devices which we use in DeterLab to launch experiments. All nodes are real physical server machines and network links represent real physical interconnects.

2.2 Security Experiments

Critical to the successful operation of our network of devices is the proxy referenced in the topology in Figure 1. The goal of our security experiments is to protect this proxy from DoS attacks. Table 1 shows configuration parameters of interest for testing DoS-resilience impact. The proxy is an especially attractive attack target because its availability is critical to the end-to-end operation of clients. The proxy is exposed to attacks spanning two protocols. Repeatedly processing many messages makes the proxy susceptible to volumetric attacks.

A well-positioned attacker, as in Figure 1, can launch a flooding attack through the proxy, which will exhaust critical proxy resources. These are critical to maintain availability and good performance from the perspective of clients [16]. Consequently, we design this proxy to maximize availability and client performance, especially in the face of an attack. We focus on managing proxy memory, latency, connections, and other critical resources in our larger security research, where we also explore topologies and configurations other than those in Figure 1 and Tables 1 and 2. In this work, we focus on instrumentation, grouping, and storage of experiments. Below we discuss the experiments' setup and measurement considerations.

2.2.1 Experiment Setup. We assume that clients are resource constrained devices (e.g., IoT devices like home security cameras, monitoring systems, and sensors) that communicate with an origin server that uses HTTP to serve content. Since clients are constrained, they cannot communicate using HTTP directly. Instead, they speak CoAP, a REST-based application-level protocol optimized for constrained machine-to-machine communication [24].

The proxy forwards CoAP to HTTP (and back) to bridge the communication gap. It receives CoAP messages from clients on its ingress, translates from CoAP to HTTP in-memory, and sends the result on its egress [5]. A typical experiment comprises multiple trials where clients send synchronously and repeatedly to the origin server, via the proxy, for a fixed period of time while an attacker runs simultaneously for a subset of that time period.

Experiment ID	Attacker	CoAP	Server Protocol	Timeout
1	N	UDP	HTTP	20 sec
2	N	DTLS	HTTP	20 sec
3	N	UDP	HTTPS	20 sec
4	N	DTLS	HTTPS	20 sec
5	Y	UDP	HTTP	20 sec
6	Y	DTLS	HTTP	20 sec
7	Y	UDP	HTTPS	20 sec
8	Y	DTLS	HTTPS	20 sec
9	N	UDP	HTTP	5 sec
10	N	DTLS	HTTP	5 sec
11	N	UDP	HTTPS	5 sec
12	N	DTLS	HTTPS	5 sec
13	Y	UDP	HTTP	5 sec
14	Y	DTLS	HTTP	5 sec
15	Y	UDP	HTTPS	5 sec
16	Y	DTLS	HTTPS	5 sec

Table 1: Sample experiments with realistic varied configuration parameters.

Experiment IDs	Description
1-4,9-12	Vary timeouts for clients without attack
5-8,13-16	Vary timeouts for clients with attack
9-16	Vary transport protocols only

Table 2: Groupings of experiments for analysis about configuration parameters. An experiment can be in multiple groupings.

2.2.2 Measurement. We choose quality-of-service (QoS) metrics at the clients which directly reflect client performance and indirectly reflect the proxy's ability to maintain liveness. We use the QoS metrics below to compare DoS-resilience of experiments with different configuration parameters:

- The distribution of client message round-trip-times (RTT).
- The number of successful and failed client requests.
- The number of retransmitted client messages.

It is insufficient to record only the client-side metrics above. For example, to diagnose *why* RTTs are slower than expected, we need knowledge about communication between devices, as well as CPU and memory profiling data. In particular, we must *trace* each message across *each hop* of the topology. This additional data is also helpful for:

(1) Computing the overhead associated with more secure but likely less performant configurations.
(2) Strengthening conclusions of improved DoS-resilience by verifying behavior from non-client devices.
(3) Identifying configuration changes that can optimize DoS-resilience.
(4) Comparing the impact of attacks on different devices in the system.
(5) Debugging our orchestration and data collection code.

3 DESIGN

In this section we describe the design of our collection, analysis, and storage framework. We break up the design into the following parts: collecting data with minimal overhead (3.1), transferring experiment data efficiently (3.2), transforming experiment data (3.3), and storing experiments (3.4). For each of these, we discuss the requirements, trade-offs, and design decisions.

3.1 Data Collection and Measurement

We want to collect the minimum amount of data necessary to validate correctness of experiments and draw conclusions about the DoS-resilience of configuration parameters. We must do so while incurring minimal instrumentation overhead in order to avoid interfering with experiment results. We are interested in measuring client QoS metrics, in addition to tracking network messages and metrics like memory and CPU utilization. Some of our client QoS metrics come from application-layer information while other are derived from network messages at the clients. We restrict our discussion to the measurement of network messages.

One approach is to use application-level logging. On high-volume devices like the proxy, attacker, and origin server, such logging is extremely resource-intensive and can redirect execution away from the application, which interferes with experiment results. Instead, we employ low-level network monitoring for the ingress and egress of each device in the experiment. Whereas in application-level logging we could choose exactly which columns to output, recording full network traces requires a separate decoding step. Correctness of experiment results is more important for us, so we choose the latter. While an experiment is running, we use the `tcpdump` utility to listen on the DeterLab experiment network interfaces of each device (not the control network), and output a `pcap` file when the experiment completes. We evaluate the overhead of `tcpdump` in Section 5.1.

3.2 Efficient Data Transfer

DeterLab resources are limited and widely shared, and storage quotas are particularly small at 10GB. Consequently, we *efficiently* transfer experiment data to a dedicated machine for processing and analysis. Network bandwidth is the limiting factor for us, so compressing the data in-flight results in better transfer performance than sending one file at a time or in parallel. The `scp` utility exposes the `-C` flag for compressing data *in-flight only* which addresses this need [26]. However, we are additionally interested in "shelving" experiments to retain the original data in a storage-efficient form. Consequently, we manually compress and decompress experiments before and after the transfer.

3.3 Transform Experiment Data

Experiment data from DeterLab comprises packet captures, metrics (e.g., CPU, memory), logs, configuration parameters, etc, and is therefore not directly amenable to analysis using the metrics referenced in Section 2.2.2. We describe in steps below how we consolidate and transform this data to the desired format.

3.3.1 Consolidating Metadata. DeterLab experiment configuration parameters are found in different locations and have different formats. For example, the NS-2 experiment topology is on a control node and the dynamic configuration in bash is on the DeterLab file system. We consolidate these into one configuration file with a single format, namely `JSON` because of its readability and ease of command-line parsing with the `jq` utility [11]. Other formats like `YAML` are good alternatives.

3.3.2 Decoding Packet Captures. We are primarily interested in decoding `HTTP(S)` and `CoAP(S)` messages. We leverage the rich community of packet dissectors in `tshark` (the command-line equivalent of `Wireshark`) for both these purposes [27, 33]. In particular, for each `HTTP` and `CoAP` message, we use `tshark` to extract the minimal set of protocol-specific options needed to conduct analysis on the experiment, which are highlighted in Table 6 in Appendix A. For messages in protocols that use transport-layer security, like `CoAPS` and `HTTPS`, we configure `tshark` to decrypt message contents through the `-o keylog_file` and `-o psk` options which take in a key-log file containing session secrets and pre-shared keys, respectively [27]. Both the key-log files and pre-shared keys are recorded and bundled in the data sent to the processing machine.

3.3.3 Transforming Data Semantics. There remain semantic issues with `tshark` output data. We detail the most relevant ones and our solutions to them below.

IP Addresses. `tshark` outputs source and destination IP addresses, but our analysis is concerned with the roles that devices play. Thus, we map IP addresses to device roles using the mapping from the consolidated configuration file in Section 3.3.1.

Message Observers. `tcpdump` does not track which device observed which messages. Thus, we name each capture with the device role that captures it.

Timestamps. We record timestamps using UNIX epoch time for simplicity, but to compare timestamps in the experiment, we only care about offsets from the initial starting time. Thus, we normalize all timestamps relative to each trial's minimum timestamp.

3.3.4 Tracking Messages. Recall from Section 2.2.2 that we must trace each message on its journey through the devices in the topology. Examples of the communication patterns of interest to us are illustrated in Figure 6 in Appendix A. These messages all correspond to the same high-level client transaction, and their creation is triggered by the first message that the client sends. But given many messages in multiple transactions, classifying messages into groups of the same transaction is challenging. This is because messages traverse many different nodes, potentially repeatedly, and a transaction's requests and responses span *different protocols*.

We assign each message an integer identifier which we call the `message_marker` such that any messages that are marked with the same `message_marker` correspond to the same transaction. For this, we use the `CoAP` `token` field, because it uniquely identifies a `CoAP` request/response pair. We must also confirm that every packet in our traces is correctly marked with this identifier. Within `CoAP` this is simple because it partitions each request/reply into `CoAP` *messages* each of which corresponds to a single `UDP` packet and contains the `token` of its `CoAP` request/reply pair. For `HTTP(S)` the `token` is concatenated with the URI and carried throughout the topology. Thus, every packet in the traces has the correct `message_marker` either in the packet header for `CoAP`-related packets or in the URI for `HTTP`-related packets, and can be collected into the appropriate transaction for detailed analysis with respect to that particular `message_marker`.

3.4 Experiment Storage

We consider two approaches to grouping experiments as in Table 2: reading transformed data directly from files, and reading indirectly from a database. The following are desirable properties and features which reduce memory consumption while sacrificing minimal performance: parallelism, redundancy elimination, selection and predicate push-downs, physical independence, query plan optimization, and loading large data from disk in RAM-size chunks. These properties are provided by most databases. We design the file-based and database-assisted approaches to have many of these properties. Nevertheless, these approaches involve significant implementation decisions and trade-offs as we show in Sections 4.3, 5.5, and 6. We expand on the two approaches below.

3.4.1 File-Based Approach. In the file-based approach, we combine experiment data from a set of experiments into a directory with the name of the supplied group. We store three files: combined configurations, metrics, and experiment results (i.e., observed messages). We store data in a column-oriented format with compression enabled, which respectively enable readers to read only the columns of interest without incurring the cost of reading an entire row (we assume rows are wide), while compression keeps the on-disk storage minimal and can improve performance. We discuss implementation and evaluation in Sections 4 and 5.

3.4.2 Database-Assisted Approach. We choose two different kinds of databases, a mature relational database and a modern OLAP database, both of which offer the properties described in Section 3.4. While the OLAP database has much better performance, we still assess the viability of a traditional relational database for grouping experiments. Among traditional databases, we compare PostgreSQL and MySQL. Though PostgreSQL is a little slower for reads, it achieves much better write performance than MySQL and supports parallel queries while MySQL does not [15, 22]. The former is critical for us as we need to bulk write data into the database, while the latter is particularly helpful for performance. Consequently, we choose PostgreSQL. Among OLAP databases, we compare ClickHouse and Apache Pinot [2, 6]. While both are open source, columnar, and support good performance, ClickHouse more closely resembles traditional database semantics like joins and primary keys. For this reason, we choose ClickHouse. We discuss trade-offs in Section 6.1.

For each experiment group, we allocate, populate, and query a *separate* database, where each database is accessible by a single server. This maintains the same read and write structure across experiment groups for both PostgreSQL and ClickHouse. Normalizing experiment data into tables and adding primary and foreign keys where relevant, we arrive at the schema in Figure 2. We choose this schema to keep tables small for performance. We achieve this by treating each primary key for a row as an identifier for its unique combination of column values, which explicitly disallows duplicate rows. The `event` and `node_metric` tables are notably not duplicate-free because they include (potentially duplicated) observations made by an observer (ref. `observer_id`). On PostgreSQL, we partition disk pages for the `event` and `node_metric` tables by the most frequently used and least selective field, the `observer_id`, in order to achieve good read query performance. For ClickHouse, tables use the `MergeTree` engine, and we store both the `event` and `node_metric` tables in order of `observer_id`). We represent each message as a template sent between devices by choosing `node_id` for the `src_id` and `dst_id` in the `message` table. Conversely, we use deployed node ID `dnid` as a reference when the node's experiment context is relevant. We discuss optimization, configuration, and evaluation of these approaches in Sections 4 and 5.

4 IMPLEMENTATION

In this section, we discuss our implementation decisions regarding the components referenced in Section 3. We make a key decision to prioritize high performance through parallelism, bulk operations, and low-level optimization via frameworks wherever possible. For flexibility, we use shell scripts to process and group experiment data. An experiment directory contains a `metadata` directory and one directory for each trial, each of which contains captures, metrics, and logs. We now detail the implementation aspects of the framework.

4.1 Compressing Data Transfer

We move all experiment files into one directory then `zip`-compress the experiment directory (the Linux `zip` command). We invoke the processing script for each experiment as separate processes. We `scp` the compressed data directory to the processing machine, and on arrival, we `unzip` the data directory for processing.

4.2 Data Transformation

We decompose implementation specifics for how we transform data into three elements, discussed here with reference to one experiment for simplicity. Note that these processing elements can take place in parallel.

4.2.1 Consolidating Metadata. We consolidate the configuration of each experiment by combining the experiment topology, DeterLab virtual and physical experiment mappings, the run-time configuration parameters, and the trials (which are each stored as a separate directory) into a single `config.json` file. We store it in the `metadata` directory of the experiment directory. The experiment topology is a network-simulator file with the `.ns` extension. From it, we parse device-to-address mappings and attack rate (which are defined using the `tb-set-ip` and `set` command respectively). The experiment information is a text-based output which summarizes machine allocations. From it, we parse the allocated hardware and operating system for each device. Finally, the dynamic experiment configuration is a bash script, which we evaluate and expose via `source`.

4.2.2 Decoding Packet Captures. We launch a separate process to decode and decrypt each packet capture using `tshark` as described in Section 3.3.2. We pass secret keys as arguments from the experiment directory. We name each input packet capture with the role of the device on which it was captured. The output of the decoding preserves this device role name so that later data transformation can programatically add the observer to the experiment results.

4.2.3 Transforming Data Semantics and Tracking Messages. We implement transformations with attention to performance and simplicity. In particular, we analyze each experiment trial in a separate process since `message_markers` are scoped to the trial level. We make use of `polars`, which is currently the most performant

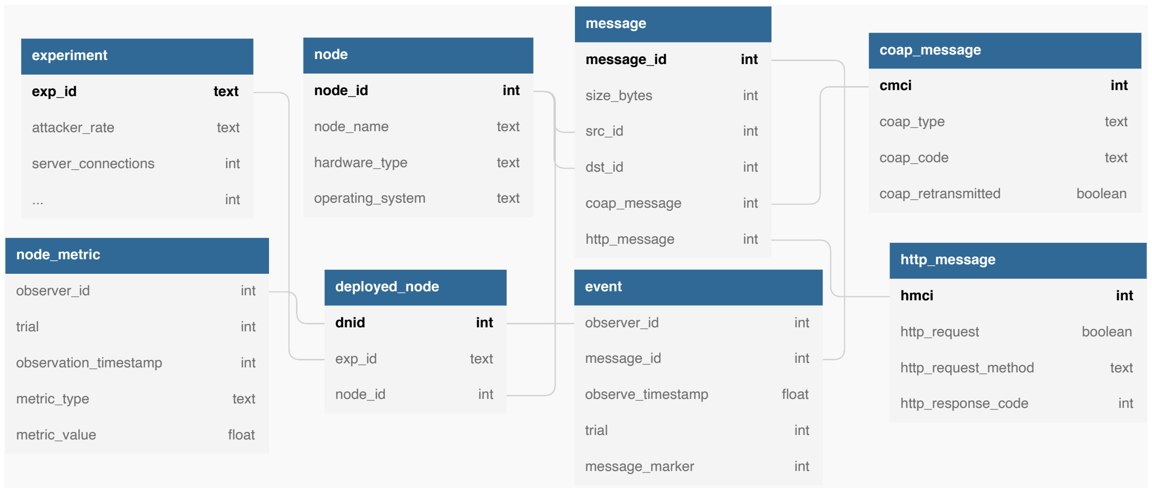

Figure 2: Diagram of the schema for the databases for grouping and analysis of experiments. Tables are normalized, redundancy-free, and partitioned where relevant.

dataframe library, to implement this processing in an "embarrassingly parallel" fashion [29]. To mark messages, we group transactions by the `token` value as in Section 3.3.4, and assign each `token` an index. This index is the `message_marker`. It starts at 1 and goes up by 1 for each transaction. We store the output using `parquet`, which uses the `arrow` columnar format, with `snappy` compression [1, 3]. `parquet` is "designed for efficient data storage and retrieval" and it provides "enhanced performance to handle complex data in bulk" [1]. We leverage it for increased read & write performance, along with reduced storage space.

4.3 Experiment Storage

We implement both file-based and database-assisted experiment grouping. In this case by making heavy use of `polars` for its fast I/O capability, since it is from 14.6 to 18.9 times faster than `pandas` [31].

4.3.1 File-Based. We represent a group of experiments as a directory of three files: configurations, metrics, and experiment results. For each file, we read the data from all experiments into a single process, make necessary modifications, and write the result to the group directory in the appropriate file. We only include configuration parameters that are relevant for analysis. To disambiguate which results and metrics come from which experiment, we add the trial and experiment identifiers as additional columns to the data before we write. The metrics file and configurations are expected to be small, while the experiment results file is expected to be quite large. This is because it contains each message from all experiments with all the columns from Table 6 in denormalized form. We write the files out using the `parquet` format with `snappy` compression to satisfy the column-oriented and compressed storage format requirements.

4.3.2 Database-Assisted. We use PostgreSQL and ClickHouse database servers with the schema shown in Figure 2 and described in Section 3.4. To load data into each database, we read all experiment and trial data into memory using `polars`. We insert experiment results into the redundancy-free tables first by scanning unique message template data and configuration data from the disk, which consumes little memory. Having obtained table IDs, we then read the subset of data for the `event` table while pushing down replacement of the newly obtained database IDs. This step can consume a substantial amount of memory, but we read a small number of columns.

Due to the number of events, the main bottleneck for this approach is inserting data into the `event` table. We use the ClickHouse driver's `insert_dataframe` method to insert `event` data in bulk from the dataframe in memory [25]. For PostgreSQL, we use the `COPY FROM` command, which "is optimized for loading large numbers of rows" [21, 23]. For even better performance, we drop the `event` table's foreign-key constraints and change it to unlogged before we `COPY`, and reinstate these afterwards [21]. Further, we configure the PostgreSQL server to choose parallel query plans when it is beneficial and optimize workers. We set `max_worker_processes`, `max_parallel_workers`, `max_parallel_workers_per_gather`, and `max_parallel_maintenance_workers` to a number slightly higher than the number of cores, 30 for us. We set `maintenance_work_mem` to a fairly large number, 1GB for us [22]. We `COPY` metrics into the `node_metric` table similarly to the `event` table. Since the `observer_id` field has few possible values and low selectivity in read queries, we improve read query performance by hash partitioning the `event` and `node_metric` tables by `observer_id`.

5 EVALUATION

In this section, we evaluate the components of the design discussed in Sections 3 and 4. For both the file-based and database-assisted experiment grouping approaches, we compare the performance and usability of their writes and reads. Throughout this section, we refer to a characteristic grouping of twelve experiments that run atop the cloud topology shown in Figure 1. This grouping is detailed in Table 3. Our dedicated data processing machine is a cloud VM running Ubuntu 20.04.2 LTS on our lab's local `OpenStack` [19] deployment.

Exp ID	Attack Rate	HTTPS	CoAPS	Uncompressed MB	Compressed MB	Compression Ratio	Messages Observed
1	100 Mbps	N	N	958	139	6.89	6,591,720
2	500 Mbps	N	N	947	137	6.91	6,423,676
3	100 Mbps	Y	N	962	395	2.44	6,365,357
4	500 Mbps	N	Y	448	144	3.11	1,730,159
5	100 Mbps	Y	Y	462	345	1.34	1,719,109
6	100 Mbps	N	Y	434	141	3.08	1,695,051
7	500 Mbps	Y	N	456	342	1.33	1,683,775
8	500 Mbps	Y	Y	435	325	1.34	1,610,971
9	0 Mbps	N	N	13	1.8	7.22	44,021
10	0 Mbps	Y	N	14	7.8	1.79	43,709
11	0 Mbps	N	Y	14	3.9	3.59	43,401
12	0 Mbps	Y	Y	15	9.7	1.55	43,161
Total	-	-	-	5,158	1991.2	-	27,994,110

Table 3: Details about a grouping of 12 low-to-moderate volume experiments which vary attack rate and protocols using the cloud topology in Figure 1. The first 8 run an attacker while the last 4 do not. These experiments involve one client sending for 100 seconds and one attacker sending for 20 seconds, for 3 trials.

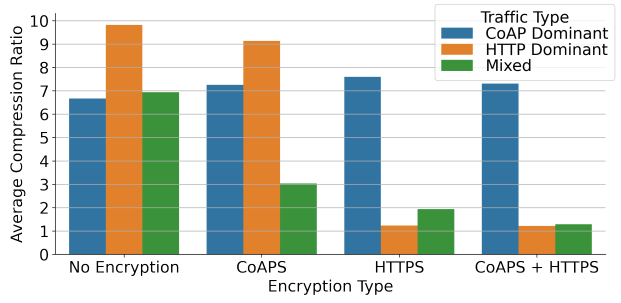

Figure 3: Bar graph comparing average compression ratio of different capture protocol distributions in various encryption scenarios. Encryption tends to lower the compression ratio.

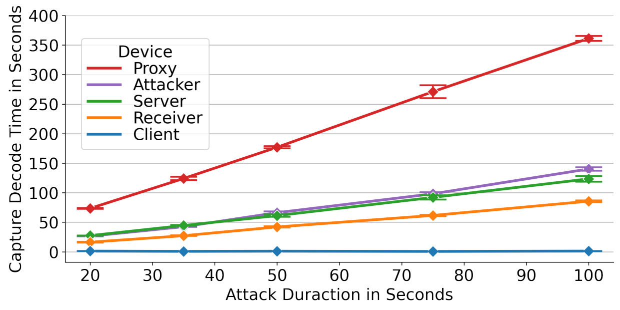

Figure 4: Line graph comparing average capture decode time versus attack duration for various capture types. There is a linear relationship between attack duration and decode time.

Volume	Proxy	Server	Attacker	Receiver	Client
High	166 sec	100 sec	128 sec	101 sec	11 sec
Medium	117 sec	87 sec	76 sec	9 sec	11 sec
Low	15 sec	13 sec	-	10 sec	11 sec

Table 4: The average decoding times in seconds for different packet captures in experiments with varying volume of traffic. There is no attacker decoding entry for low volume since low volume experiments do not involve attack traffic.

Table Name	Row Count	PgSQL Storage	CH Storage
`event`	27,994,110	1,394 MB	207.1 MB
`node_metric`	45,935	4,552 kB	135.7 kB
`message`	99	24 kB	1.29 kB
`deployed_node`	60	32 kB	945 B
`experiment`	12	32 kB	3.41 kB
`coap_message`	6	32 kB	357 B
`node`	5	32 kB	382 B
`http_message`	2	32 kB	569 B
Total	28,040,229	1,399 MB	207 MB

Table 5: Table sizes and row counts for the experiments from Table 3 grouped in PostgreSQL (PgSQL) and ClickHouse (CH). CH consumes significantly less storage than PgSQL.

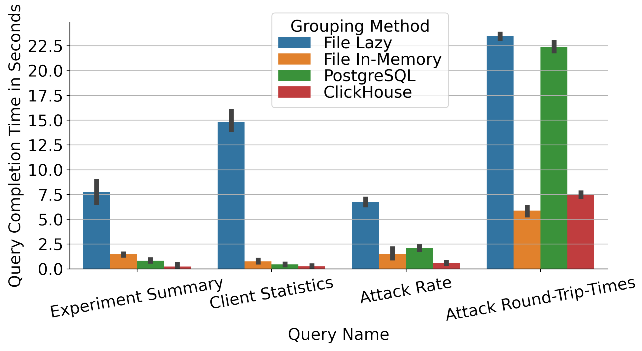

Figure 5: Bar graph comparing the average completion times of four representative single-experiment read queries, while varying the experiment grouping method. ClickHouse mostly completes faster than file in-memory.

The machine has 24 CPU cores of type Intel(R) Xeon(R) CPU E5-2699 v4, running at 2.20GHz. It has 48GB of RAM, realized via three 16GB DIMM RAM chips, and a 300GB volume backing a `ext4` file system.

5.1 Measuring Messages

We verify that CPU and memory utilization are not heavily affected by logging packets via `tcpdump`. To do this, we compare experiments with the highest attack rate, 1Gb/s, with attacks up to 100 seconds, as we expect them to have the highest overhead. In experiments with a 50 second attack, running without `tcpdump` produces 3,596 messages with an average of 68.59 requests per second (RPS), while using `tcpdump` produces 3,601 messages with an average of 68.49 RPS. With an attack of 100 seconds, running without `tcpdump` produces 2,685 messages with an average of 92.07 RPS, while using `tcpdump` produces 2,670 messages with an average of 92.88 RPS.

At the origin server, the CPU overhead of `tcpdump`-logging is 0.66% on average and 2.7% in the worst case, while the memory overhead is 7.5MB on average and 11.66MB in the worst case. At the proxy, CPU overhead is at worst 0.9%, while worst case memory overhead is 66MB. We assign the proxy and origin server relatively powerful hardware on DeterLab, each having 8 cores and 16GB of RAM. Consequently, we find this overhead minimal and acceptable.

5.2 Compressed Data Transfer

We have found that compressing experiment data is highly beneficial for transfer performance. From Table 3, we observe that `zip` compresses experiments with a median compression ratio of 2.76 and average compression ratio of 3.38. Interestingly, the maximum compression ratio of 7.22 (and other close ratios) corresponds to experiments that send network traffic in plain-text. Conversely, the minimum compression ratio of 1.33 (and other close ratios) corresponds to experiments with `DTLS` enabled for `CoAP` and `TLS` enabled for `HTTPS`.

Figure 3 reports the average compression ratio for individual packet captures, and shows that encryption tends to lower compression ratios. This agrees with the idea that securely encrypted content should be indistinguishable from random bits. `CoAP`-dominant capture compression ratios are unaffected because we make use of a single pre-shared key for `DTLS` encryption. We have empirically observed that these worst-case compression ratios only add a few more seconds to the end-to-end transfer. Even in this worst case, compressed transfer is 33% faster than uncompressed transfer. With network bandwidth as our bottleneck, we find compressing the data in-flight highly beneficial.

5.3 Decoding Messages

Each experiment in the grouping from Table 3 has multiple trials and clients, and each produces a capture for each device in each trial. This results in 168 different packet captures to process. Packet captures from experiments with higher volume take longer to process, ranging from 4.01s for those with no attacker, up to 177.81s for those with the strongest attacks. This is the most expensive step in the data processing and storage framework, but because we decode each packet capture in its own process, the processing machine is able to run relatively efficient analysis in parallel. In particular, it processes all 168 packet captures in 2 minutes and 52 seconds. If the processing were sequential instead, it would take the machine approximately 17 minutes.

Because the proxy sees each message multiple times and speaks multiple protocols, its packet capture is the most computationally expensive to analyze, as in Table 4. In fact, for medium and high-volume attacks, the proxy capture decoding time dominates the overall decoding times. The difference is less pronounced for low-volume attacks; proxy captures decode in 15 seconds on average, while the server and client decode in 15 and 11 seconds respectively. For medium and high-volume attacks, where proxy decode times are respectively 117s and 166s, the second most expensive capture decoding times are 87s and 128s (74% and 77% of the proxy decoding time) respectively.

We evaluate the scalability of capture decoding in Figure 4, where we run the maximum attack rate of 1Gb/s on DeterLab while increasing the duration of the attack. We observe that there is a linear relationship between average decode time and attack volume. The proxy has a larger constant of proportionality since it sees twice the amount of messages as other devices. For the heaviest attack and the largest capture, the average decoding time is 6 minutes. We address this in Section 6.

5.4 Transforming Data

Transforming and combining packet capture data is relatively fast when using `polars`. The minimum transformation time is 9.4s, the median is 17.8s, the average is 19.6s, and the maximum is 36.0s. The fastest transformations correspond to the lowest-volume experiments while the slowest transformations correspond to the highest-volume experiments. Running the transformations in parallel is again highly beneficial, taking a total of 40 seconds to complete across all experiments. Compared to the sequential completion time of 3 minutes and 55 seconds, running transformations in parallel completes in a sixth of the time.

5.5 Experiment Storage

We evaluate the time and storage to group processed experiments together for both the file-based and database-assisted approaches.

5.5.1 File-Based. Grouping the experiments from Table 3 with the file-based approach completes with a rate of 3.96 seconds per GB, totalling 19 seconds to complete. On completion, the three files in the group's directory, which are stored as `snappy`-compressed `parquet` files, occupy a total of 903.2MB of storage space. Most of the storage comes from the 903MB-sized `results.parquet` file. `metrics.parquet` occupies 224KB and `configurations.parquet` occupies 8KB of storage. The resultant grouping occupies less than half the storage of compressed experiments before processing, and nearly a sixth of the uncompressed version. At its peak, file-based grouping uses 7.69GB of RAM. This approach is not likely to scale when experiments jointly occupy more disk space than available RAM. We discuss this in Section 6.

5.5.2 Database-Assisted. The PostgreSQL database-assisted approach takes an average of 2 minutes and 9 seconds to group the experiments referenced in Table 3. 69% of the time (90s) is spent executing the `COPY` command on the `event` table. 19% of the time

(24.5s) is spent running ANALYZE on the database to optimize later read queries, 10% of the time (13s) is spent reading experiment data into memory and massaging it, and the remaining 2% (2.5s) is spent setting up the database before data insertion.

The COPY phase is the largest bottleneck in grouping. This is slower than the file-based approach by a factor of 6.8. Interestingly, the average and peak memory usage are 14% and 74% more than the file-based approach, totalling 8.75GB and 13.35GB respectively. The higher average memory usage comes from materializing data in-memory before COPYing, while the higher peak memory usage likely comes from PostgreSQL's lazy garbage collection policy during the COPY phase.

ClickHouse takes an average of 24.9 seconds to group the experiments in Table 3. 14% of the time (3.68s) is spent reading and formatting event data in memory, and 41% of the time (10.37s) is spent sending event data to the database. 39% of the time (9.75s) is spent reading and filtering message data, while the remaining 6% of time (1.5s) is spent reading data in memory and populating the remaining tables. Impressively, the ClickHouse grouping completes in a fifth of the PostgreSQL grouping time, and is only 30% slower than the file-based approach. The peak memory usage, 8.74GB, is comparable with PostgreSQL's average of 8.75GB and consumes two thirds of PostgreSQL peak's memory usage. ClickHouse grouping consumes 14% more memory than the file-based approach.

Table 5 shows database table sizes after grouping. The event table is the dominant storage factor, holding nearly 28 million rows. The total storage size is 1,399MB for PostgreSQL, which is respectively 70% and 27% of the storage required for the compressed and uncompressed pre-transformed data. This is 35.45% more storage than the file-based approach. The ClickHouse total storage is 207MB, which is respectively 4% and 10% of the storage required for the compressed and uncompressed pre-transformed data. This is 22.9% of the storage required for the file-based approach.

5.6 Query Performance

We now compare average read query performance for file-based and database-assisted groupings. For database-assisted grouping, we compare PostgreSQL and ClickHouse with equivalent schemas, while for file-based grouping, we compare lazy and in-memory reads. In the in-memory model, we execute every query on the full experiment data memory-resident after reading it from disk once at startup. In the lazy model, we interleave disk reads with computation pushdown for every query using the polars Lazy API [30]. We compare queries that are computationally expensive, using four representative queries, as shown in Figure 5. Numeric completion times for each query are wall-clock times from Jupyter notebooks which we average over five trials. We restart database server and notebook processes before each query.

The first query summarizes the number of messages and active time for each node in every trial and experiment. The lazy file read method is the slowest, taking 7.76s to complete. In-memory and PostgreSQL reads respectively take 1.45s and 824ms to complete. ClickHouse, the fastest, completes in 220ms. This summary query produces only max and min aggregations over the event table's observe_timestamp column, which databases and parquet files traditionally maintain to different extents. These results show that database reads take better advantage of these statistics.

The second query enumerates every client message's send time, receive time, response code, and round-trip-time, along with request-per-second statistics. The approaches' completion times are ordered just as in the first query: lazy file reading completes in 14.8s, while in-memory, PostgreSQL, and ClickHouse reads respectively complete in 753ms, 446ms, and 253ms. This query involves multiple joins and filters, rendering it quite complex for a disk-based read to complete efficiently.

The third query measures the effective attack rate as a function of time. Lazy file reading completes in 6.74s, while in-memory, PostgreSQL, and ClickHouse reads respectively complete in 1.47s, 2.12s, and 575ms. Note that the completion time order of in-memory and PostgreSQL reads is swapped for this query. The aggregations in this query involve a distinct count and a sum, where the former is more difficult to maintain in a DBMS than in-memory. We conjecture that this is why, for this query, in-memory reads slightly outperform PostgreSQL reads, completing 31% faster.

The fourth and final query is a summary of the round-trip-time along each hop for each attack message, all the way from the attacker to the receiver. Lazy file reading completes in 23.5s, while in-memory, PostgreSQL, and ClickHouse reads respectively complete in 5.86s, 22.4s, and 7.44s. This query is the only one where in-memory outperforms ClickHouse, completing 21% faster. We attribute this difference to the large number of output records which this query selects. In particular, it is faster to bulk-modify and stream a large number of records column-wise and in-memory than it is to read this data from disk first.

In summary, ClickHouse reads significantly outperform PostgreSQL reads among database-assisted groupings, while in-memory reads significantly outperform lazy file reads among file-based groupings. ClickHouse outperforms optimized in-memory reads with polars for queries which produce few output records, while queries which produce many output records benefit from memory-resident data, thus avoiding the cost of disk reads. Nevertheless, ClickHouse, like most databases, regulates its own memory usage, resulting in little risk of running out of memory. This is in stark contrast to reading data with polars, where we have hit the out-of-memory limit many times as we conducted our research.

6 DISCUSSION

In this section we discuss the performance (6.1), usability (6.2), and scalability (6.3) of the file-based and database-assisted approaches to storing experiment groups, and other stages of experiment analysis.

6.1 Performance

We are able to measure network events without instrumentation overhead, at the cost of noticeably slow processing, with the two largest bottlenecks being packet decoding and insertion into the database. The cost of decoding packet captures is discussed in Section 5.3. A first approach to improving decoding performance would be to partition large packet captures and process them in parallel, then combine the resulting outputs at the end. Even better, tcpdump supports the option to rotate output capture files using the -G and -C flags [28]. The decoding step can further benefit from high performance packet capture decoders like [20] and [12], but which support decryption and decoding `CoAP`. Alternatively, it is possible to capture packets with `tshark` directly. Our preliminary experiments found that this has high overhead, but future work could make this a viable option.

The second biggest bottleneck is the time to insert data into PostgreSQL, despite using the `COPY` command. To our knowledge, there is no substantially faster way of loading this data into PostgreSQL. We've shown that ClickHouse is an excellent alternative to PostgreSQL, providing comparable write performance and better read performance than the in-memory file-based approach [6].

6.2 Usability

Regarding the usability of cryptographic secrets, we store pre-shared keys and key-log files in plaintext since we are building an academic prototype. In a realistic setting, one shouldn't leak keys or key-log files. A good option can be to embed decryption secrets into the `pcapng` packet capture format, which enables this capability [32].

We also observe three usability implications for the file-based and database-assisted groups. First, as the size of data approaches the available RAM, the file-based approach to grouping will run out of memory, making this processing framework less usable. However, the database will regulate the amount of RAM that it uses and won't be killed by the operating system. Second, SQL queries are simpler to write and easier to read than hand-crafted `polars` or `pandas` queries. We found that the traditional PostgreSQL SQL dialect does not seamlessly translate to ClickHouse SQL, missing key convenience features like the `RETURNING` clause and joining multiple tables inline to avoid clunky pairwise joins [7]. Finally, formalism in databases for structuring data combined with the many decades of experience of building databases results in the database-assisted grouping approach in general, and PostgreSQL in particular, being less error prone.

6.3 Scalability

For both the file-based and database-assisted grouping approaches, the loading phase involves reading experiment data into memory. If the total size exceeds RAM, then we need to read data in memory-sized batches, perhaps using the semantics of `ChunkedArray` from `Arrow` [3]. If the experiment data exceeds the size of the disk (e.g., through dozens of long duration Gb/s attacks), then we need a distributed storage solution.

For a database approach, we recommend Apache Pinot or Firebolt instead of ClickHouse, since distributed databases are core to their design [2, 6, 14]. Configuring, loading, and querying these databases will not drastically differ from our PostgreSQL approach. However, we expect the file-based approach to need more major structural change to accommodate a distributed file system, such as setting up a distributed cluster like Hadoop, and coming up with a data layout plan that provides good read performance. A distributed cluster likely offers even better performance than our single-database approach, but we are interested in the design space of computation on a single machine as it more closely matches the constraints of our security research.

7 CONCLUSION

In this paper, we've proposed and evaluated methods for measuring, processing, and storing groups of experiments to study the effect of various system configuration parameters. The measurement process relies heavily on network message monitoring, realized via `tcpdump`. While we've shown it to incur little-to-no instrumentation overhead, there is a significant trade-off in packet capture processing performance. We've shown that processing data in bulk, in parallel, and with lazy computation are highly beneficial.

Query read performance is generally better assuming database-assisted grouping due to the normalization and partitioning schemes we choose, but writing a group of experiments is 6.8 times slower for PostgreSQL. ClickHouse is tolerably 30% slower. Despite this, the database approach manages RAM more carefully and exposes a more user-friendly interface for querying data than hand-crafted programmatic queries. In addition, databases support certain features of interest that the file-based approach does not, like multi-user authentication, access control, and stricter semantics for transactions.

While these approaches have served us well for moderately sized experiment groupings, around 10GB, the approaches likely have to change for much larger experiment groupings. In particular, at much larger scale, the database approach we evaluated in this paper likely translates more smoothly to high-performance distributed OLAP databases like Apache Pinot or Firebolt [2, 14], while the file-based grouping will likely need more structural change when moving to a distributed file system. We recommend ClickHouse for experiment groupings whose sizes are between our moderately-sized tens of gigabytes and much larger, terabytes-sized ones, that require a distributed storage solution.

We believe that our ideas about experiment data measurement and storage can be beneficial for other users of DeterLab, especially if they run high-volume experiments. We encourage the DeterLab community to assess how to integrate these ideas within the existing infrastructure to support broader experimentation use cases. Noting the limitation of the MongoDB cluster which DeterLab maintainers currently provide, we recommend that maintainers increase configuration access and experiment storage methods for users. To facilitate reproducibility and extensibility, we have made our code, scripts, and both raw and processed data publicly available through the `git` repository in [13], along with instructions to reproduce our results.

ACKNOWLEDGMENTS

The authors wish to acknowledge Henry Corrigan-Gibbs for interesting discussions on generalizing takeaways of the work.

REFERENCES

[1] Apache. 2022. Apache Parquet. https://parquet.apache.org/
[2] Apache. 2022. Apache Pinot™: Realtime distributed OLAP datastore | Apache Pinot™. https://pinot.apache.org/
[3] Apache. 2022. Format. https://arrow.apache.org/overview/
[4] Terry Benzel. 2011. The Science of Cyber Security Experimentation: The DETER Project. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '11)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2076732.2076752
[5] Angelo P. Castellani, Salvatore Loreto, Akbar Rahman, Thomas Fossati, and Esko Dijk. 2017. Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP). RFC 8075. https://doi.org/10.17487/RFC8075

[6] ClickHouse. 2022. ClickHouse - Fast Open-Source OLAP DBMS. https://clickhouse.com/
[7] ClickHouse. 2022. Why ClickHouse didn't allow more than one JOIN in query? · Issue #873 · ClickHouse/ClickHouse. https://github.com/ClickHouse/ClickHouse/issues/873
[8] DeterLab. 2022. MAGI Configuration - DETERLab Documentation. https://docs.deterlab.net/orchestrator/orchestrator-config/#dbdl-configure-the-magi-data-management-layer
[9] DeterLab. 2022. Orchestrator Data Management - DETERLab Documentation. https://docs.deterlab.net/orchestrator/data-management/
[10] DeterLab. 2022. Orchestrator Quickstart - DETERLab Documentation. https://docs.deterlab.net/orchestrator/orchestrator-quickstart/
[11] Stephen Dolan. 2022. jq. https://stedolan.github.io/jq/
[12] Ted Dunning. 2022. pcap-filter. https://github.com/tdunning/pcap-filter original-date: 2016-04-08T01:54:13Z.
[13] Amir Farhat. 2022. CSET DoS Flooding Paper Code and Data. https://github.com/amirfarhat/cset-dos-flooding
[14] Firebolt. 2022. Cloud Data Warehouse For Engineers | Firebolt. https://www.firebolt.io/
[15] Krasimir Hristozov. 2022. MySQL vs PostgreSQL – Choose the Right Database for Your Project. https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres
[16] Jelena Mirkovic, Alefiya Hussain, Brett Wilson, Sonia Fahmy, Peter Reiher, Roshan Thomas, Wei-Min Yao, and Stephen Schwab. 2007. Towards User-Centric Metrics for Denial-of-Service Measurement. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (San Diego, California) *(ExpCS '07)*. Association for Computing Machinery, New York, NY, USA, 8–es. https://doi.org/10.1145/1281700.1281708
[17] MongoDB. 2022. MongoDB: The Application Data Platform | MongoDB. https://www.mongodb.com/
[18] UK NCA. 2022. DDoS attacks are illegal - National Crime Agency. https://www.nationalcrimeagency.gov.uk/?view=article&id=243:ddos-attacks-are-illegal&catid=2
[19] OpenStack. 2022. Open source cloud computing infrastructure. https://www.openstack.org/
[20] PcapPlusPlus. 2022. PcapPlusPlus. https://pcapplusplus.github.io/
[21] PostgreSQL. 2022. 14.4. Populating a Database. https://www.postgresql.org/docs/14/populate.html
[22] PostgreSQL. 2022. Chapter 15. Parallel Query. https://www.postgresql.org/docs/14/parallel-query.html
[23] PostgreSQL. 2022. COPY. https://www.postgresql.org/docs/14/sql-copy.html
[24] Zach Shelby, Klaus Hartke, and Carsten Bormann. 2014. The Constrained Application Protocol (CoAP). RFC 7252. https://doi.org/10.17487/RFC7252
[25] Marilyn System. 2022. Features — clickhouse-driver 0.2.4 documentation. https://clickhouse-driver.readthedocs.io/en/latest/features.html#numpy-pandas-support
[26] Tatu Ylonen Timo Rinne. 2022. scp(1): secure copy - Linux man page. https://linux.die.net/man/1/scp
[27] tshark. 2022. tshark. https://www.wireshark.org/docs/man-pages/tshark.html
[28] Craig Leres Van Jacobson and Steven McCanne. 2022. tcpdump(1) man page | TCPDUMP & LIBPCAP. https://www.tcpdump.org/manpages/tcpdump.1.html
[29] Ritchie Vink. 2022. Introduction - Polars - User Guide. https://pola-rs.github.io/polars-book/user-guide/
[30] Ritchie Vink. 2022. Lazy API - Polars - User Guide. https://pola-rs.github.io/polars-book/user-guide/optimizations/lazy/intro.html
[31] Ritchie Vink. 2022. Polars, lightning-fast DataFrame library. https://www.pola.rs/
[32] WireShark. 2022. TLS. https://wiki.wireshark.org/TLS#embedding-decryption-secrets-in-a-pcapng-file
[33] Wireshark. 2022. wireshark. https://www.wireshark.org

A TABLES AND FIGURES

tshark Field ID	Description
`_ws.col.Time`	Packet UNIX epoch observation timestamp.
`_ws.col.Source`	Packet source IP address.
`_ws.col.Destination`	Packet destination IP address.
`_ws.col.Protocol`	Protocol that sent this packet.
`_ws.col.Length`	Packet size in bytes.
`coap.type`	The CoAP type of the packet.
`coap.retransmitted`	Flag indicating if this packet is a CoAP retransmission.
`coap.code`	The CoAP code of the packet.
`coap.mid`	The CoAP message ID of the packet.
`coap.token`	The CoAP token of the packet.
`coap.opt.proxy_uri`	The URI which the proxy should forward the request to.
`http.request`	Flag indicating if this is an HTTP request.
`http.request.method`	The method of this http request (e.g., GET, POST, etc).
`http.request.full_uri`	The URI which this request requests.
`http.response.code`	The response code attached to this HTTP response.
`http.response_for.uri`	The URI of the request that generated this response.

Table 6: This table contains all the fields we decode from packet captures with tshark.

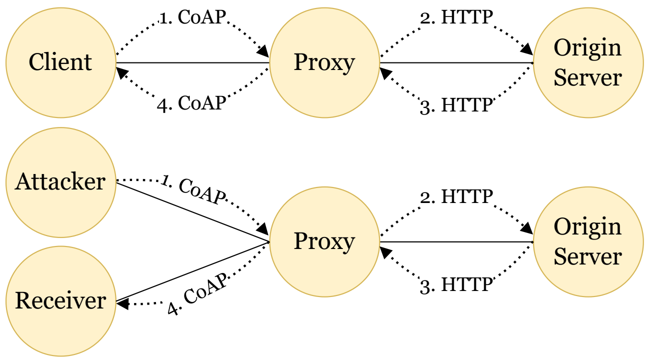

Figure 6: Diagrams of client and attacker protocol communication sequences. Identifying each message's originating transaction is non-trivial.