Critique on the paper: Generating representative, live network traffic out of millions of code repositories

Abhijit Deo
ENI, Bits Pilani Goa campus
2019A8PS0041G
f20190041@goa.bits-pilani.ac.in

Keywords— traffic generation, paper-critique, network analysis

I. Problem Statement

Generating a substantial amount of representative application traffic is a challenging task that can be approached in two ways: by replaying packet traces or by using traffic generators. However, both methods have limitations in terms of generating truly representative traffic. Packet trace replaying provides realistic traffic patterns but does not account for application behavior or network conditions. Publicly available traces also have low throughput, making them unsuitable for stress testing. On the other hand, traffic generators generate traffic according to distributions, which may not capture complex application logic. Therefore, the research question is whether it is possible to develop a system that can generate live network traffic in significant amounts, incorporating various application logic to ensure representativeness

II. Solution

The authors of the paper have proposed a system to generate live traffic that is more realistic with the help of real-world applications. They call it Dynamo (DYNAmic Mass Orchestration).

DYNAMO operates in three phases: an offline phase for finding suitable open-source projects, a bootstrapping phase for selecting projects and preparing virtual interfaces, and a traffic generation phase where DYNAMO populates interfaces with live application traffic.

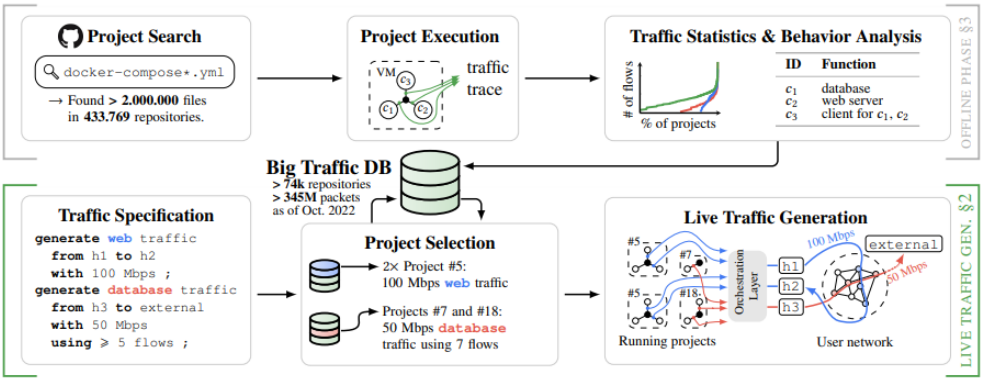

Figure 1: DYNAMO leverages the abundance of open-source projects to build the Big Traffic database. Based on a user's traffic specification, DYNAMO then finds and orchestrates adequate open-source projects for live traffic generation.

The offline phase is executed only once, which builds a database that contains traffic stats and metadata about each open-source project. The live traffic generation part is of an on-demand type where the user can give the specifications of the traffic required, and the DYNAMO will query this through its database for appropriate projects, and then the system orchestrates the traffic with the help of selected projects. As open-source projects are not built with the intent to keep generating traffic at a specific constant rate, DYNAMO has a sliding window control mechanism that maintains the throughput.

Because the DYNAMO database has the metadata of the traffic generated in the offline phase, it is able to generate complex traffic patterns like having specific expected interpacket arrival times, burstiness of traffic, or even matching custom traffic distributions and also able to have behavior like client-servers.

III. Evaluations

As DYNAMO first needs to execute the open-source projects in the offline phase, it is necessary to find projects from diverse domains that could be run in a container environment with a high probability of generating traffic. The authors used GitHub to find such open-source projects. With the help of GitHub API, authors found 433k repositories with docker files, and authors considered 67% of the projects for this experiment i.e. 293k, out of which they could execute around 38k repositories. Reasons behind failure for other projects vary from source code bugs, no longer available containers, and insufficient memory. These 38k repositories generated various kinds of traffic, as shown.

Table 2: The ten most-observed destination ports belong to diverse applications such as web, database, or Bitcoin.

port	description	port	description
5001	iPerf and IPFS	8333	Bitcoin traffic
443	HTTPS traffic	55606	Dynamic/private port
5672	RabbitMQ traffic	27017	MongoDB traffic
4001	IPFS (file system)	6099	Selenium (browser tool)
80	HTTP traffic	1433	MySQL traffic

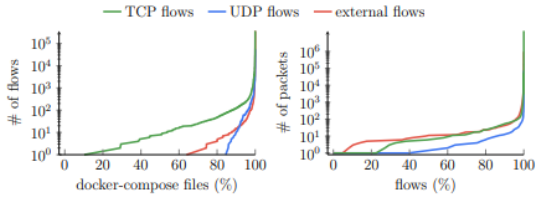

Figure 3: Half of the applications generate more than ten flows, while the median flow size is around ten packets.

Authors further analyzed the data and showed the diverse nature of the packets generated, 33% constitutes for HTTP/S, database, and Bitcoin traffic, while the remaining packets were accounted by DNS, Ethereum, telnet, ssh, etc. The authors also noted the large amount of traffic generated by network-related repositories like HTTP stress tests and Integration tests for some projects. 89% of the traces generated contained at least one TCP flow and 16% UDP flow. Generally, these UDPs performed DNS queries in the database. There was one public destination IP for at least 38% of files, i.e., outside the VM environment.

The traces generated during the projects are also uploaded on zenodo server hence this can act as a database for other projects as well.

IV. Related Work

There are some already existing works that try to generate the network traffic with the software or hardware or replaying the packets from an environment. But these methods lack the real-world properties of the network. Some of these works are TCPreplay, trex etc

Also, there are some publically available datasets of the traces, like CAIDA (Anonymized Internet Dataset) or MAWI. However, these databases don't consider network conditions and generally have low throughput. There are other research works like using data-mining workloads with the help of minimalistic data centers, the idea which was proposed in pFabric. This idea could generate complex traffic, but the few in real-life behaviors like client-server won't be feasible.

V. Shortcomings

The process of collecting the GitHub datasets to analyze the traces for the offline phase is limited by the GitHub APIs and is time taking. Also, with this comes the challenges of storing the GitHub repositories. One would require significant storage to save all the GitHub files. Also, large amounts of GitHub repositories were of no use; one of the reasons for this was no longer available containers or bugs in code. This could have been avoided/reduced by having looked at the status of CI/CD pipelines of the respective projects and could have saved time and other resources. One more way to optimize the offline process would be to look at memory footprints from the CI/CD pipelines and consider whether to go ahead with that repository or not, as one of the reasons for docker failure was memory shortage. There are some other shortcomings as well, but authors have proposed ways to tackle them in the future, like scraping the readme files to follow the correct dockerization procedures or including test suits of the libraries etc.

With the above methods, one could reduce the overhead while creating the database.

Also, extending the project for other data repositories like GitLab, and considering few other Container solutions, project could have more impact.