École Polytechnique Fédérale de Lausanne

Magma: A Ground-Truth Fuzzing Benchmark
Remote Fuzzing Campaigns

by Benedek Hauer - benedek.hauer@epfl.ch

Bachelor Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Ahmad Hazimeh, EPFL
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 7, 2022

Contents

1 Introduction

Magma is a ground-truth fuzzing benchmark that includes a library of real targets, such as libpng, libtiff, libsndfile, etc... with previous real bugs that have been ported into them based on outdated CVE reports. By inserting canaries into the programs, fuzzing campaigns in magma can detect which bugs have been triggered, therefore based on this evaluation and introducing more of these bugs into Magma, benchmarking becomes more accurate.

2 Problem Statement

As jobs can be very resource-demanding and long in terms of running-time, in the case of having a lot of jobs to run on a local machine, we could have a set of campaigns executing for several days. In order to improve this bottleneck from the point of view of speed, resource management and workload distribution, my work on Magma during this semester consisted of dispatching campaigns across eventual remote machines in order to save local resources and improve parallelism. This could be made possible with the use of Beanstalkd.

3 Beanstalkd

3.1 What is Beanstalkd?

Beanstalkd is a work queue that was originally designed for reducing the latency of page views in high-volume web applications. As its interface is very generic, it could be adapted to fit our use case. Beanstalkd works with the usage of communication tubes (queue channels) that can be produced into or consumed from by all connected nodes. A Beanstalkd message (more commonly referred to as a ”job”) is of the form of a String. Only one client node can reserve (consume) a job at a given time, which will guarantee us that no unspecified duplicate jobs will get done uselessly. An exploitable advantage, is that one machine/node can set up multiple clients in parallel, each serving a different purpose. This is useful to avoid tube entangling when multiple threads are willing to use the same client at the same time while trying to work with different tubes, but all this will be addressed later. In short, Beanstalkd is a queue (or set of queues) that can be produced into and consumed from, on a global scale.

In order to communicate with its work queue, Beanstalkd provides many client libraries, such as Greenstalk, which is usable with the help of Python. It was my first choice, as it is well-documented, and stable.

3.2 Use Case in Magma

Firstly, we have to think of what constitutes a ”job” in Magma, i.e. what level of granularity we need. As magma evaluates fuzzers on targets with specific programs, the most intuitive answer is to simply have on tuple of (fuzzer, target, program, args, fuzzargs) per job. This lets us exploit

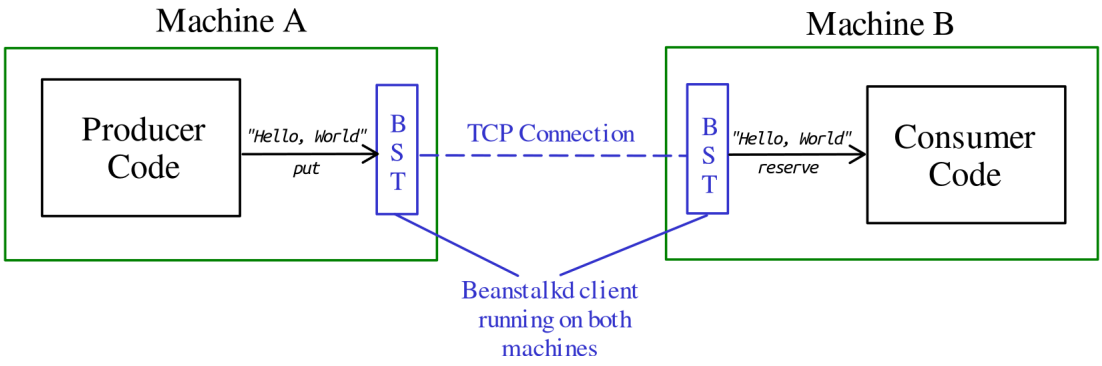

Figure 1: Beanstalkd instance running on two different machines

parallelism when running campaigns, as well as gives us a clear understanding of what really happens under the hood. Although, besides needing jobs to run, remote machines need to be configured in order to know which CPU layout to use, which fuzzing parameters to be set, etc.... Therefore we also need to transfer configuration instructions over tubes.

4 Communication Tubes

In the big picture, we would ideally like one dispatcher sending out the jobs, and multiple remote machines consuming and executing them, before sending the results back. This leads us to two trivial tubes: `jobs` and `feedback`. As we want to transport as few configuration jobs as possible, we would ideally need a `broadcast` tube as well, which contains setup information for campaigns to run. Lastly, worker machines should also send negative feedback back to the dispatcher in case a job fails, which is done through the `graveyard` tube. The dispatcher will then have three clients: one responsible for sending broadcasts (configurations), one for the jobs, and one that collects feedback from both the `feedback` and `graveyard` tubes. The worker side will use one client per tube.

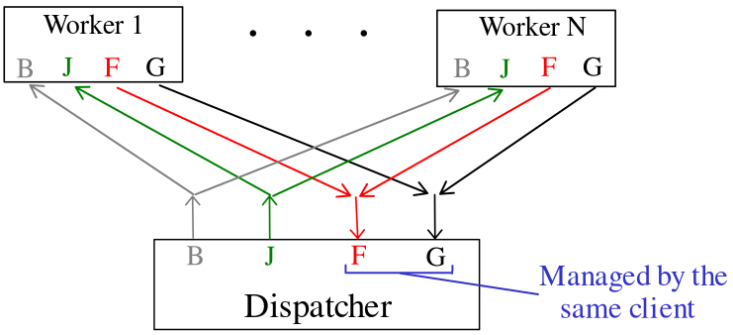

Figure 2: Beanstalkd Tubes

Broadcast Tube Peeking

The implementation for the `broadcast` tube went over many changes. In the worker machine configuration (discussed later), the workers have knowledge of the dispatcher's IP to connect to it, but the number and identity of the workers are initially unknown to the dispatcher.

The first implementation of the `broadcast` tube was relying on the fact that workers would first set up a tube named with their IP address, announce it to the dispatcher on a global channel called `default`, which, upon client initialization, is the only existing Beanstalkd tube by default. The dispatcher would then set up a tube for each connection and broadcast messages over all of these tubes. However, if many workers are connected, then a lot of redundant information will be sent, and optimally we would not want the dispatcher to have knowledge on how many (and which) workers are connected, as it should only care about producing jobs and consuming results.

The second implementation relied on the fact that workers would `reserve` (consume) jobs from the `broadcast` tube, and release them into the tube whenever they read the information. This approach has been quickly discarded due to the fact that it could lead to starvation and useless resource usage, as the workers could reserve and release the job more than once. Moreover, if a worker dies during the processing phase of the broadcast, the information would never get released, which would block the whole network.

The final solution to the broadcast problem, is *peeking*: Workers would only peek into the `broadcast` tube without actually reserving the jobs, which will not waste time and resources, nor will it lead to a global deadlock. **Note:** As tubes are FIFO, only the oldest broadcast could be read, which is stale. As Beanstalkd does not offer LIFO alternatives, the `broadcast` tube is actually a single-slot queue that the dispatcher modifies when a new configuration is needed to be set.

As a conclusion, the difference between the `broadcast` tube and the other tubes is that these other tubes are to be both consumed from (`jobs` for the workers, `feedback` and `graveyard` for the dispatcher), and produced to (`jobs` for the dispatcher, `feedback` and `graveyard` for the workers), whereas the `broadcast` tube is not to be consumed from, but only peeked into by the workers, in order for them to read the configuration instructions when they connect to the network at any point in time.

As a brief summary on the execution pattern: The dispatcher broadcasts the general configuration, puts jobs into the tubes, waits until results or dead jobs are available, and fetches them. The worker, on the other hand, peeks into the `broadcast` tube, starts fetching jobs, and begins execution. If an error is encountered or the job finishes, results or dead jobs are communicated back to the dispatcher.

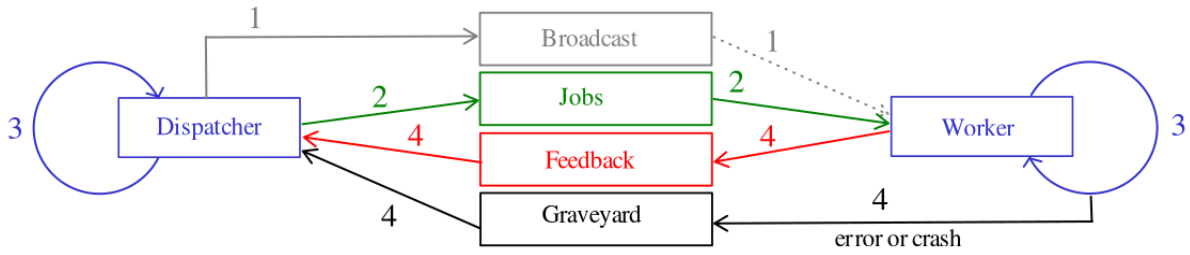

Figure 3: Broadly Summarized Execution Pattern

5 Pre-run Configuration

In order to set up the dispatcher and worker machines, as well as to configure jobs to run with specific parameters, we need pre-configurations. These pre-configurations - in addition to the previously-used `captainrc` that needed alterations to fit our use case - include information on connection and data-transfer. My first thought about configuration files was to have one for the dispatcher and one for the worker, written following the Bash syntax. This was a feasable idea, however, as my implementation is mostly done in Python due to Greenstalk, reading Bash-syntaxed files would have been quite difficult and non-intuitive, not to mention it might have introduced a lot of bugs into the parsing process. Now, as the idea of having only bash-syntaxed configuration files has been discarded, I decided to use the `JSON` format as it is easily parsable using Python, which led to having one configuration file for the dispatcher and one for the worker. However, I quickly realized that it is not best to write everything in Python using only `JSON` files for configuration files, as shell scripts come in most handy when it comes to starting programs, doing file system-related operations, remotely copying files or doing some useful pre-run checks on a lower (but more concise) level. Eventually, I found the most suitable solution, which is to use three configuration files: Two for the dispatcher (basic configuration and jobs configuration) and one basic configuration file for the worker.
Note: All files and folders are relative to `captain/`.

5.1 Dispatcher Configuration

The dispatcher configuration file (`dispatcher/dispatcherconfig`) contains the following attributes:

TEMP_RESULTS
WORKDIR
COPY_TYPE
JOBS_CONFIG

where:

1. TEMP_RESULTS is a directory that contains all completed job folders before they are all merged together into the
2. WORKDIR directory.
3. COPY_TYPE is one of {scp, rsync}, which is the protocol used for fetching remote jobs from the workers. It is recommended to use rsync though, as it is significantly faster than scp.
4. JOBS_CONFIG is the JSON configuration file that contains all information on jobs to give to the workers.

Note 1: TEMP_RESULTS and WORKDIR need to not exist prior to launching the main dispatcher script, as it would lead to data loss and inconsistent results otherwise. **Note 2**: There are two more attributes in dispatcher/dispatcherconfig, called WORKDIRS and KEEP_WORKDIRS, that are actually worker attributes and have been added later when creating the quickRun.sh script to launch a local run without the need of starting both the main dispatcher and worker scripts on the same machine (same role as the previous run.sh). I will explain what these two attributes are used for when discussing the worker part.

5.2 Campaign and Jobs Configuration

The jobs configuration file (named jobsConfig.hjson) contains the same attributes as the previous captainrc, except it is written in JSON format instead of being Bash-syntaxed. Note the hjson file name extension. Hjson is a user interface for JSON that enforces less syntax constraints and supports comments, whereas regular JSON does not. Luckily, there is a hjson software package for Python. Although performing slightly worse than the basic json, it is an acceptable price to pay in exchange for user-friendliness.

5.3 Worker Configuration

The worker configuration file (workerconfig) contains the following attributes:

WORKDIRS
KEEP_WORKDIRS
DISPATCHER_IP
WORKER_IP

where:

1. WORKDIRS is a directory that contains all WORKDIRs for worker threads per worker machine. It seems counter intuitive at first, because one machine could use only one WORKDIR, as there seems to be no difference between a worker machine running a set of campaigns and what the previous run.sh did. However, the difference resides in the *feedback* stage of the execution. As we want to exploit resources as much as possible, when a worker thread finishes computing, instead of exiting and idly waiting for all the other campaigns to complete on that machine, it should inform the dispatcher straightaway that the result is ready to be fetched, so that it can start working on the next available job as soon as possible.

Therefore per-thread workdirs need to be independent from one another, and WORKDIRS simply acts as a container for all of them.

2. KEEP_WORKDIRS is a value that, if set to 1, will make sure that worker machines keep the WORKDIRS directory after all campaigns are finished, instead of cleaning them up after the dispatcher closes the connection. If KEEP_WORKDIRS is set to 0 or unset, the worker machine delete the directory. The use of this attribute is for later checking what happened on the worker side if inconsistencies happen. For example, after a series of jobs fail, the dispatcher could relaunch the campaigns and later check what went wrong on the worker side.

3. DISPATCHER_IP is the IP address of the dispatcher machine, needed by the worker machine to set up the Beanstalkd connection.

4. WORKER_IP is the IP address of the worker machine, communicated at the *feedback* stage in order for the dispatcher to fetch the results using `rsync` or `scp`.

This concludes the configuration part, the next sections will address the execution parts of the projet.

6 Preprocessing

6.1 Dispatcher Side

When launching the main dispatcher script (`dispatcher/runDispatcher.sh`), we first go through two preprocessing stages. The first stage is very brief, and it checks that:

1. The required Python dependencies are installed (Greenstalk and Hjson)

2. `dispatcher/dispatcherconfig` has been filled in well, meaning

   (a) No required attributes are left unset,

   (b) WORKDIR and TEMP_RESULTS don't exist before their creation,

   (c) WORKDIR and TEMP_RESULTS are not the same (compare their respective absolute paths), as TEMP_RESULTS is deleted after all the jobs are fetched and copied into WORKDIR. For the same reason, WORKDIR being a subdirectory of TEMP_RESULTS is not accepted either.

   (d) An instance of Beanstalkd is not currently running, as it would lead to desynchronization between the dispatcher and the workers, which will result in all jobs being graveyarded. Note: the user can still bypass this check with the `-f` flag passed to `runDispatcher.sh`, if they consider the warning to be a false alert.

During the second stage, we perform checks on the jobs configuration file to make sure everything seems in order at first glance. This includes the following:

1. No required attributes are missing,

2. Attributes match one of their possible values (for example `logical`, `physical` and `socket` are the only acceptable values for WORKER_MODE),

3. Attributes that need to be integers or lists are type-checked,
4. All fuzzers in `FUZZERS` are in `magma/fuzzers/`,
5. All targets in `TARGETS` and `OVERRIDDEN_TARGETS` are in `magma/targets/`,
6. All programs specified for a specific target `TARGET` exist in `magma/targets/TARGET/configrc.hjson`'s `programs` object.

With these two preprocessing stages, we ensure that most jobs configurations that pass these checks are consistent.

6.2 Worker Side

On the worker side, we do similar checks for attributes in `workerconfig`, namely:

1. Greenstalk and Hjson are installed.
2. `WORKDIRS` and `KEEP_WORKDIRS` are both set.
3. `WORKDIRS` does not exist. If it does, it means that it did not get properly deleted after a previous job, and might be of use to the user for checking out what happened.

7 Setup

In this short section of our execution process, all machines create the directories where the campaigns' data will be stored and start the beanstalk daemon after cloning its repository (except on the machine where a worker program is running alongside the dispatcher). After setting up the clients for each tube, the dispatcher has to first clear the `feedback` tube. This prevents it from starting off with a desynchronization with the workers. It can happen when the main dispatcher program unexpectedly exits (due to a `SIGINT` for example) before fetching all the results. In this case workers (having no knowledge about this) would still provide feedback. This would lead to fetching results that correspond to stale jobs that have been dispatched in previous runs.

8 Execution - Dispatcher Side

The execution process of the code ran on the dispatcher's side can be divided into two main parts: (1) Broadcasting and populating the `jobs` tube, and (2) fetching the results from the `feedback` and `graveyard` tubes.

8.1 Populating the Tubes

The broadcasting part consists of stringifying the global configuration file (which can be easily done with `Hjson`), and inserting it into the `broadcast` tube. This `broadcast()` function returns a `configId` that needs to be prepended to all outgoing jobs, for workers to check that they are following orders according to the correct configuration file.

For the jobs part, the dispatcher selects the needed configuration fields, namely `REPEAT`, `FUZZERS`, `TARGETS`, `OVERRIDDEN_TARGETS`, `PROGRAMS` and `FUZZARGS`, and creates a list of jobs of the form `"repeatID,fuzzer,target,programName programArgs,fuzzargs"`, where `repeatID` is a number ranging from 0 to `REPEAT` -1 (both inclusive). `repeatID` might seem redundant, as the job is put `REPEAT` times into the `jobs` tube anyways, however the worker machines will use this ID for creating the unique subdirectories of `fuzzer/target/program/` in the archived campaign results to make sure no directory names overlap across worker machines scattered all around the network. The parsing phase follows the following steps, to repeat for each fuzzer in `FUZZERS` (note: `jobs` is initially an empty list):

1. Generate a list that contains `"fuzzer,target"` string tuples, where `target` is either fetched from the `OVERRIDDEN_TARGETS` if specified, or from `TARGETS` otherwise.
2. For each element in the list, if `PROGRAMS["fuzzer,target"]` exists, we fetch the arguments of these specified programs and add `"programName programArgs"` to each element of the list, otherwise we execute this step for all programs in target's `configrc.hjson` and flatten out the fetched program lists by distributing it over `"fuzzer,target"` in order to produce `"fuzzer,target,programName programArgs"` elements.
3. To all these string tuples, append an extra slot containing either the empty string (if no fuzzargs have been specified for `"fuzzer, target"`), or `FUZZARGS["fuzzer,target"]`.
4. Add this new list to `jobs` `REPEAT` times after prepending every string tuple with the unique job's `repeatID` ranging from 0 to `repeatID-1`.

After having gone through these steps for all fuzzers, the parser passes on `jobs` onto the main dispatcher program, which can then populate the `jobs` tube.

My first implementation did only interpret the `REPEAT` parameter during the parsing process, and did not pass the repeat ID back to the dispatcher. This resulted in directories getting overwritten when more than one machines were set up as workers. Putting the `repeatID` into the job tuples as well ensures that every created folder is indeed unique in the network.

Example A configuration file set up like this:

```
REPEAT: 2
FUZZERS : ["afl", "honggfuzz"]
TARGETS : ["libtiff"]
OVERRIDEN_TARGETS : {
    "afl": ["libpng"] # Ignores TARGETS
}
PROGRAMS: {
    "honggfuzz": ["tiffcp"] # Ignores tiff_read_rgba_fuzzer
}
FUZZARGS {
```

```
    "afl,libpng": "-x /magma_shared/png.dict"
}
```

will produce the following jobs into the `jobs` tube:

```
"0,afl,libpng,libpng_read_fuzzer,-x /magma_shared/png.dict"
"1,afl,libpng,libpng_read_fuzzer,-x /magma_shared/png.dict"
"0,honggfuzz,libtiff,tiffcp -M @@ tmp.out," # tiffcp's arguments are automatically appended to it
"honggfuzz,libtiff,tiffcp -M @@ tmp.out,"
```

which will all be prefixed by `"<configId>,"`, which is the configuration id returned by the previously-called `broadcast()` function. Note: `FUZZARGS` is only specified for `"afl,libpng"`, therefore Honggfuzz jobs will end with a comma as no fuzzargs were provided.

8.2 Fetching Results

After the dispatcher is done populating the `jobs` tube, it will poll the `feedback` and `graveyard` tubes and fetch incoming results. The format of results passed through the `feedback` tube is `"isLocal,username,workerip,path_to_file"`, where

1. `isLocal` is an integer that stands for whether the worker that published this result was on the same machine as the dispatcher or not. If yes, the dispatcher will only do a simple copy instead of using `rsync` or `scp`.
2. `username` is the name of the user on the worker machine (`$USER`).
3. `workerip` is the IP address of the worker (set up in `worker/workerconfig` on worker machines)
4. `path_to_file` is the location to copy from.

If the job is not local, the dispatcher will execute a remote copy from the location `username@workerip:path_to_file` into `TEMP_RESULTS`. An element of the `graveyard` tube is of the same format as jobs inserted into `jobs`, as the worker that failed to execute that particular job simply sends it back. When the dispatcher finishes fetching all the jobs, it does some post-processing work and cleanup, but these parts will be addressed later.

The result fetching process went through a few changes before I decided to use `rsync/scp`. Initially, it was an implementation that relied on setting up an HTTP server on the dispatcher's side, and the workers would the do a `POST` request to it. The issue I encountered, was that handling `POST` requests was not implemented in Python's HTTP server, which meant that it was my task to write it. Given the fact that HTTP `GET`s and `POST`s are not designed for folders nor for compressed content, I decided to drop the idea of using an HTTP server, and went with the `rsync/scp` approach, which also provides good security and doesn't require explicitly setting up a server beforehand.

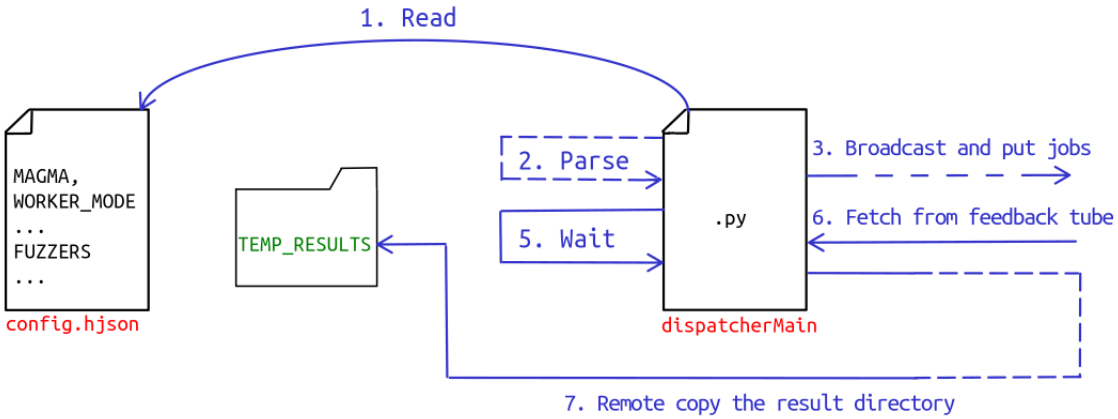

Figure 4: Main Dispatcher Execution Process (`dispatcherMain.py`)

9 Execution - Worker Side

The execution process on the worker side is divided into multiple steps:

9.1 Setup

After setting up the clients for each tube, we read (interpret) the latest broadcast and create:

1. A ThreadPool that contains worker threads that will run jobs.
2. A queue of sets of CPUs for threads to pop from. This queue is created by a helper function, that reads the output of `lscpu -bp`, and populates the queue with sets of CPUs, following to the `WORKER_MODE`, `WORKERS`, `WORKER_POOL` and `CAMPAIGN_WORKERS` parameters. About the `FUZZER_CAMPAIGN_WORKERS` parameter: Each thread, will pop CPUs from the queue until they have at least `FUZZER_CAMPAIGN_WORKERS` logical cores at their disposal. `FUZZER_CAMPAIGN_WORKERS` is by default set to `CAMPAIGN_WORKERS` (which is defaulted to 1 in case it is unspecified).

   Implementation Details: When processing these parameters, compromises need to be made, which cannot be set as preferences by the user, as they might have no knowledge of the capabilities of machines they are working with. Therefore, the following points are enforced:

   (a) As `WORKERS` corresponds to the minimum total number of logical cores to be allocated, we will round up the number of logical workers in order not to split up CPU nodes (by default a logical node contains one logical CPU). Therefore, in logical mode, as logical CPUs are the smallest unit, the user will be guaranteed to have exactly `WORKERS` logical CPUs that will be used. However, in physical and socket mode, there might be additional CPUs, for example here, in red, we cannot break up physical nodes apart, and in green we cannot split up the socket even if `WORKERS=1`.

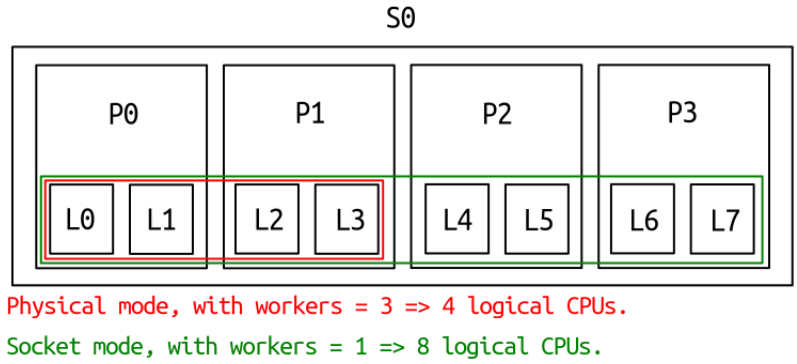

Figure 5: `WORKER_MODE` example output in physical and socket mode

(b) If `WORKER_POOL` is set, then we have to enforce logical mode. As each remote machine has a different type of layout (w.r.t. indices of logical CPUs), the program cannot enforce a particular list of CPUs to use if the user wants to use physical or socket mode. If we did not make this choice, and consider the physical case (it is the same for socket mode), then this means that either: (1) Few/None of the physical CPUs would be used if we decide to drop all CPUs that are not in a pair with another CPU in `WORKER_POOL`. This would minimize the resources to be used by jobs, and would leave many resources for the personal use of the user. Or, (2) Many/All of the physical CPUs would be used, if we decide to keep all physical CPUs that contain at least one element present in `WORKER_POOL`. This would maximize the resources to be used by jobs, and could leave no resources for the personal use of the user. In both cases, we could be massively cutting down on resources (either for the user, or for the job), therefore, if `WORKER_POOL` is set in the configuration file, we will enforce logical mode.

(c) `CAMPAIGN_WORKERS` works similar to how the `WORKERS` parameter is handled, meaning it will not split up CPU nodes as it acts as a lower bound on the number of logical workers to use per campaign.

(d) `WORKERS and WORKER_POOL` cannot be set at the same time, as they could give contradicting instructions to the broadcast interpreter. This requirement is checked in the dispatcher part of the program, during the preprocessing phase.

3. A set of `job_id, job_body` tuples to keep track of currently running jobs. This information is used in case of an error, the main worker thread will graveyard all the elements in this set and empty it.

As we need to build docker images sequentially, we need a lock to make sure the builds are synchronized. Initially, this lock was not part of the code, as builds were running on the worker's main thread (therefore no concurrency issues would arise), however I later decided to build docker images on worker threads. The reason for this decision, is that each thread owns a `workdir_jobId` (unique per job). Even though the main threads knows which directories need to be created during the process of job-fetching, I thought it would make more sense not to divide the tasks of building and running the campaign among two different threads.

9.2 Fetch Jobs

On each worker machine, the main worker thread is tasked with continuously fetching the jobs from the `jobs` tube and then inserting them into a generator containing jobs. However, between these two steps it needs to check if the configuration ID of the jobs match the ID returned when they fetched the broadcast in the Setup stage. (Note: Job IDs are consistent across the network, therefore if the dispatcher's `broadcast()` function returns `123`, then the same broadcast read by a worker will be guaranteed to have ID `123`.) If the IDs match, the execution goes forward, otherwise the client re-reads the broadcast, re-completes the Setup step if the IDs now match, otherwise it graveyards the current running jobs and stays idle until the dispatcher closes the connection.

Note: An element that I needed to add towards the end of the project, was an attribute that accounts for the number of free threads in the pool. As the main worker thread fetches jobs continuously, it would continue fetching jobs even if all its worker threads are busy. In the case of multiple worker machines in the network, it would lead to starvation, as the machine that connects first will completely deplete the `jobs`, forcing the other machines to stay idle until the Beanstalkd connection is closed.

9.3 Make Threads Work

Worker threads go through five main steps:

1. Fetch a job from the job generator that the main thread continuously populates. This job has a `jobId` that is used to create the unique `workdir` for that particular job. These unique workdirs are named `workdir_jobId/`, and reside in the `WORKDIRS` directory.

2. Get the sets of CPUs which they will be bound to. These sets they fetch from the queue will be used to run the job they have been given in part 1. If no `FUZZARGS` are given for that particular job, then the number of sets fetched will always be 1.

3. Build the docker image for the campaign. This step is done by a helper script called `worker/imageBuilder.py`, which first creates the `log/` directory in `workdir_jobId/` then redirects the output of `build.sh` into `log/fuzzer_target_build_WORKERIP.log`. The `WORKERIP` part is useful to distinguish between different machines' build outputs for same pairs of `(fuzzer,target)`.

4. Run the campaign. `run.sh` has been slightly modified to fit our use case. It now only runs one campaign of `fuzzer, target, programName programArgs, fuzzargs`. The `lock/` directory has been removed as locking is now done when necessary in the Python scripts and the interpretation of parameters related to CPUs have been moved there as well. `get_unique_cid` has also been renamed and changed to create the directory with the (globally unique) `repeatId` name in `ar/fuzzer/target/program/`.

5. Send feedback to the dispatcher and repeat steps 1 through 5.

10 Postprocessing and Cleanup

After all jobs are executed and fetched, both the dispatcher and the workers need to go through a small postprocessing and cleaup stage.

1. Dispatcher side: When all the results are fetched, the dispatcher has to merge all directories that have been previously fetched from the workers. At this point, everything is in `TEMP_RESULTS`, which contains `workdir_jobId/` directories. With the help of the `rsync` command, the dispatcher merges all these folders together into `WORKDIR`, and removes the `TEMP_RESULTS` directory as it is of no use anymore. It can now close the Beanstalkd connection (by killing the Beanstalkd process), and delete the `beanstalkd/` directory.

2. Worker side: When all the results are fetched, the worker deletes `WORKDIRS` if `KEEP_WORKDIRS` is set to `1`, kills the beanstalkd process and deletes the `beanstalkd/` directory before exiting.

Note: It was necessary to add a file called `run_post_extract.py`, which parses the configuration file and runs `post_extract.sh`. As `captainrc` does not exist anymore, this file addition was needed to parse the `hjson` configuration file.

This concludes the code execution part of the project.

11 Performance

Thanks to the delegation of tasks to remote machines, a huge speed increase should happen when multiple workers are connected compared to local runs. I ran a few simple performance measurements that show the speed improvement of tasks. As I did not have access to more than two machines to run campaigns, the results will only be shown for one and two workers. I also included results comparing the time difference between this version of Magma and the previous one (in terms of a single run). The runs have been performed with `socket` worker mode, for `(afl, libpng)` with a `REPEAT` parameter of 2. The docker images were already built on both machines, prior to the measurements.

TIMEOUT	Local Run (New Version)	Local Run (Previous Version)	One Worker (Non-Local)	Two Workers (One Local, One Remote)
10m	1209s	1202s	1218s	617s
5m	609s	602s	615s	316s
1m	127s	122s	135s	77s
30s	69s	62s	76s	45s
5s	20s	12s	27s	16s

Figure 6: Performance comparison between local and non-local runs

We can observe that setting up the Beanstalkd connection and dispatching jobs introduces an

overhead of approximately seven seconds locally compared to the previous version of Magma. This overhead is even greater when the jobs to run need to be done on a remote machine. It is however expected as the goal of this project was to support non-local jobs as well as local ones, and the dispatcher has no initial knowledge on which jobs will be ran locally (if any at all). Although, we can see a tremendous improvement when running two worker machines: the execution time is almost halved, which is also to be expected as we can now exploit two CPU sockets. **Note:** Even though the shorter `TIMEOUT` is, the greater our communication overhead impacts the running-time, as fuzzing campaigns are usually longer-time jobs, this overhead becomes less significant.

12 Challenges and Difficulties

Due to the fact that my project had been given few strict constraints for the final goal as it mostly relied on my implementation of the feature, it was inevitable to face a few challenges throughout the semester. The main ones were the following:

1. Setting up the Beanstalkd connection between remote machines: Even though the solution to this task was very simple, I had some difficulties figuring out where the connection errors were coming from. As it was not entirely clear in the beginning, that the daemon has to be started on both machines before using greenstalk to remotely connect to it, it took me quite some time to figure out.

2. Broadcasting: Although broadcasting seems like a simple task, finding the correct and reliable way to implement it was a bit difficult. As the program hangs when peeking is wrongly used, debugging was not easy. Eventually, after understanding the documentation and getting simple examples to work, I managed to adapt it to our use case without any major issues.

3. How threads should be given tasks: As I was not fully familiar with using concurrent data structures and thread pools prior to this project, I encountered a considerable challenge when I got to the part where I had to implement how threads should be given jobs. I was first thinking about an approach where one thread would be created per job fetch, but my supervisor talked me out of it and provided me with pseudo-code about the main implementation idea, which proved to be a very clean solution to the problem.

4. Tube clients: Before using one client per cube, I encountered many concurrency issues when I got to the part where I had to implement the thread tasks. My implementation was initially using one Greenstalk client per worker machine, which then lead to non-deterministic execution with wrong messages being put into wrong tubes. I first tried a locking approach that I dropped very early on, as it would make the code very difficult to understand. It was when I tried to use multiple clients per machine that this problem got instantly solved with this simple solution.

13 Possible Improvements

I believe that three improvements could have been made with more time to work on the project, and they were my next priorities when I got to the end of this semester.

1. Crash Handling: Crashes are not handled well. Cleaning up is done to a certain extent, but I had only started thinking about crashes towards the end of the project, therefore I could not handle all the exceptions or errors everywhere. For example, if a SIGINT happens in specific parts of the program, Beanstalkd is not stopped, which will throw an error message during the next run's preprocessing stage, requiring the user to kill the Beanstalkd process and relaunch the batch. Crashes on the worker machines are not well-handled either. The same error is shown, however in case of a crash during the program, the dispatcher will wait forever as the `feedback` and `graveyard` tubes will not be populated by workers that have crashed.

2. Graveyarding improvement: Graveyarding should not only occur when the configuration IDs don't match. It should also occur throughout the whole execution. The reason does not necessarily have to be a crash or an error. For example, if a worker machine does not have enough resources (number of logical cores) to complete a job, instead of sitting on it indefinitely, it could send it back to the dispatcher, letting it know what the issue was. The dispatcher could then dispatch it again for other machines to fetch. Of course if there is only one worker machine, the process should be stopped if the job comes back every time it is dispatched.

3. Better logs: On worker machines, working threads are sometimes crossing over each other's outputs, which can confuse the user. For example, when the user is prompted to enter their password for (un)mounting tmpfs, new information can appear on the screen due to other threads executing, which does not make it explicit for the user that they are required to input something.

4. The post extraction part could have been done in a better way, by refactoring and using only one script instead of two.

5. `quickRun.sh` can be improved to remove the overhead of setting up a Beanstalkd connection, as the jobs will only be ran locally hence no tube communications are required.

14 Future Work

Firstly, the first two improvement would be necessary for a reliable workflow. Not only would they be beneficial in terms of user experience, but they would also give the program more resilience to crashes or configuration errors.

Concerning the back-end part of the project, as jobs are dispatched remotely, one improvement on a longer term would be introducing communication between workers, meaning workers

could relay jobs to other worker machines in the network if they do not have the necessary resources to complete a job. They could even save building time by relaying a job to a particular machine, if it already possesses a `fuzzer/target` docker image for a specific job.

On a front-end part, if the access to some dedicated machines is given, using a web user interface for generating jobs and observing progress would also be an improvement from the point of view of user experience.

15 Conclusion

Overall, with the help of this long execution pipeline, the initial problem statement has been addressed. Although improvements can be made, I believe that my main goal for this project has been reached. It is now possible to run remote jobs on Magma, which will be of great benefit both in terms of speed and saving local resources. However, I think of this project more of as a first building block towards the improvement of running remote campaigns with Magma, than as a simple ”done” project. Even though I faced a few challenges here and there throughout the semester, I had a lot of fun working on it, as I got to discover topics I was previously not very familiar with, such as fuzzing, work queues, docker images and containers, bash scripts and even Magma itself. On a final note, I would like to thank Ahmad Hazimeh, my supervisor, who guided me throughout this project in order to help me achieve my goal.

16 References

Beanstalkd (https://beanstalkd.github.io/)
Greenstalk (https://greenstalk.readthedocs.io/en/stable/)