BNWEngine Version Difference

Teh Min Suan

October 2024

1 BNWEngineV4: Key Optimizations

The optimizations and improvements made in version 4 can largely be attributed to three key areas: the implementation of multiprocessing, a new cache design, and optimized code logic.

1.1 Multiprocessing

Before diving into how multiprocessing is implemented in `BNWEngineV4`, let's first explain the Global Interpreter Lock (GIL) in Python. The GIL is a mutex that controls access to Python objects, ensuring that only one thread executes at a time. This lock is necessary because CPythons memory management is not thread-safe. While the GIL makes single-threaded programs efficient and simplifies the integration of C extensions, it can become a bottleneck in multi-threaded programs, particularly those that are CPU-bound. To achieve parallel execution in such cases, we introduce multiprocessing in `BNWEngineV4` using Pythons `multiprocessing` library.

```python
import multiprocessing as mp
```

The `multiprocessing` library enables the spawning of multiple processes to run tasks in parallel in Python. For our purposes, we create a pool of processes, with the number of processes set to {`mp.cpu_count()`.

```python
PROCESSOR_COUNT = mp.cpu_count()
pool = mp.Pool(processes=PROCESSOR_COUNT)
```

Once the pool is created, tasks can be passed to it, and the library will automatically distribute the tasks across the processes based on their workloads. In `BNWEngineV4`, we will submit three types of tasks to the process pool, as shown below.

```
for i in range(PROCESSOR_COUNT):
    pool.apply_async(BNWWorker.initialization, (i,))
```

```
for _ in range(PROCESSOR_COUNT):
    pool.apply_async(BNWWorker.checkTableUsage)
```

```
pool.apply_async(BNWWorker.run_riskmodel, (alert_information, alert_data,))
```

`BNWWorker.initialization` is primarily used to set up log files for each `BNWWorker` that will perform risk score calculations and updates. It also initializes variables for MongoDB collections, including `bayesian_features`, `risk score_compiler`, `test_timeline`, and `alerts`. Lastly, it retrieves the `user_or_server` and `system_type` from `bayesian_features` for use in calculating the risk score.

On the other hand, `BNWWorker.checkTableUsage` is run daily to monitor the usage of score tables in alerts. If a score table hasn't been used for the current day, it clears the table's content to conserve memory. When needed, `BNWWorker` will query the database to retrieve relevant data and store it in the corresponding score table.

The `BNWWorker.run_riskmodel` function is the core of the risk score calculation and update process for alerts. Whenever an alert message is sent to `BNWEngineV4` via RabbitMQ, it processes the alert message and, along with relevant cached alert data, passes it to `BNWWorker` for risk score calculation and updating.

1.2 New Cache Design

As mentioned in previous section, whenever `BNWWorker.run_riskmodel` is called, the relevant alert data will be passed to the pool of processes from a cache. The cache is completely revamped to enable faster extraction of relevant data and storing lesser amount of data. Previously, `BNWEngineV2` will get all the past 30 days alert data from the database and store the data directly in a list. Now, `BNWEngineV4` will pull the data with the following pipeline:

```
pipeline = [
    {
        '$match': {
            '_timestamp': {'$gte': epochYesday},
            'false_positive': 0,
            'type' : { '$exists': True }
        }
```

```
    },
    {
        '$sort': {'_timestamp': -1}
    },
    {
        '$group': {
            '_id': '$type',
            'alert': {'$first': '$$ROOT'}
        }
    },
    {
        '$project': {
            '_id': 1,
            'alert': {
                '_timestamp': 1,
                'type': 1,
                '_id': 1,
                'attacker_ip': 1,
                'user_name': 1,
                'dst_role': 1,
                'dst_ip': 1
            }
        }
    }
]
result = list(coll_alerts.aggregate(pipeline))
```

Here's what each stage of the pipeline does:

1. `$match` stage:
   This stage filters documents based on specific criteria:
   - `_timestamp`: The document's timestamp must be greater than or equal to `epoch30Days`, which represents the timestamp for 30 days ago.
   - `false_positive`: This field must be 0, meaning only non-false-positive alerts are selected.
   - `type`: This field must exist in the document (`$exists: True`), so only documents with a defined type are considered.
2. `$sort` stage:
   The matched documents are sorted by `_timestamp` in descending order (-1). This ensures that the most recent alerts are prioritized.
3. `$group` stage:
   The documents are grouped by the `type` field:

- _id: This is set to $type, meaning each group corresponds to a unique type.
- alert: For each group, the most recent alert (from the sorted results) is selected using '$first': '$$ROOT'. Here, $$ROOT represents the entire document.

4. $project stage:
   This stage defines which fields to include in the final output:

   - _id: The grouping key (the type value) is kept.
   - alert: A sub-document is created that includes the following fields:
     - _timestamp: The timestamp of the alert.
     - type: The alert type.
     - _id: The alerts unique identifier.
     - attacker_ip: The IP address of the attacker.
     - user_name: The username involved in the alert.
     - dst_role: The destination role.
     - dst_ip: The destination IP address.

The full data pull mentioned earlier occurs only once a day or when the cache size exceeds CACHE_SIZE.

```
CACHE_SIZE = 100 * 1024 * 1024
```

When BNWEngineV4 receives an alert message from RabbitMQ, it updates its cache using the following pipeline.

```
filterParam = {
    "_id": {"$gt": last_ObjID},
    "false_positive": 0,
    'type' : { '$exists': True }
    }
projectParam = {
    '_timestamp': 1,
    'type': 1,
    '_id': 1,
    'attacker_ip': 1,
    'user_name': 1,
    'dst_role': 1,
    'dst_ip': 1
    }
result = list(coll_alerts.find(filterParam, projectParam).sort([('_timestamp',
↪   1)]))
```

Regardless of which pipeline is used for pulling data from the database, the data will be stored in the cache in the following format:

- `alertData`: The cache that holds alert data from the database.
- `entity_name`: The name of the entity responsible for the alert.
- `alert_type`: The type of the alert.
- `alert_information`: The data retrieved from the database, filtered using projections in the pipeline.

```
alertData =
{
    entity_name:
    {
        alert_type: alert_information
    }
}
```

Whenever `BNWEngineV4` needs to retrieve relevant data from the cache to pass to `BNWWorker`, it will use the entity name and send `alertData[entity_name]` to `BNWWorker`. If an alert involves multiple entity names, all related alert data will be passed to `BNWWorker`.

1.3 Optimized Code

Most of the optimizations won't be detailed here, as they involve changing entire blocks of code, and there are too many to track individually. While these optimizations wont be discussed, they account for the majority of the performance improvements achieved by refining the code. This section will focus on the major revamps and refactoring of the code.

Previously, in `BNWEngineV2`, upon receiving an alert message from RabbitMQ, an internal function called `run_riskModel_v2()` was triggered. This function performed several checks and processing steps before calling `riskmodel_v2.run_riskModel_v2()`. After obtaining the result, it would invoke `UpdateScoreInsertTimeline_v2()` to update the risk score in the timeline.

In `BNWEngineV4`, all of this is handled within `BNWWorker`. Additionally, there is now only a single `run_riskModel()` function, which performs the necessary checks and processing before calling `riskmodel()` to calculate the risk score.

Moreover, two functions from `risk_score_compilerV2.py` have been relocated. The `decay_adjustedriskscore()` function has been moved to `BNWEngineV4.py`, and the `update_riskscore()` function to `BNWWorker.py`. The former runs daily to decay the risk score calculated by `BNWWorker`, while the latter updates the calculated risk score in the database.

2 Results

In this section, we present the test results of `BNWEngineV4.py`, focusing on its correctness and performance in comparison to `BNWEngineV2.py`. The output of `BNWEngineV4.py` is deemed correct if it matches the output of `BNWEngineV2.py`, with the exception of minor differences in timestamps or update times.

Key points for this section:

- RabbitMQ as a Non-Bottleneck:
  RabbitMQ is not a bottleneck in alert processing. All tests begin only after the RabbitMQ queue in BNWEngine is fully populated with alerts.
- Initial Cache State:
  The cache of `BNWEngine` starts off empty and is cleared before each test.
- Timer Start/Stop:
  The timer starts once the first output from `BNWEngine` is updated in the database and stops after X correct updates are made, where X represents the total number of alerts to be processed. Therefore, the recorded time reflects the processing duration for $X - 1$ alerts. Note that this excludes the startup time of `BNWEngine`, which benefits `BNWEngineV2`, as it has a significantly longer startup time compared to `BNWEngineV4`.
- Alert Processing Routes:
  - Route A:
    Alert information is sent to `listen_log`, prompting the `alerts_worker` to update the alerts collection in the database. This causes `BNWEngine` to retrieve and cache the alert data from the database.
  - Route B:
    Alert information is sent directly to RabbitMQ, bypassing the `alerts_worker`. This means the alert will not be updated in the database, and `BNWEngine` will not cache the information.
- Entity Name Handling:
  Alerts with repeated entity names are processed via Route A, while alerts with randomized entity names are processed via Route B.
- System Load:
  No other applications or scripts that could significantly impact machine performance are running during the tests.
- Test Environment:
  The tests are conducted on an Ubuntu machine with 8 logical cores and 16 GB of RAM.

2.1 Correctness of `BNWEngineV4`

To validate the behavior of `BNWEngineV4`, its output is compared with that of `BNWEngineV2`. The outputs are considered correct if the data being updated or inserted into the `test_timeline` and `riskscore_compiler` collections are identical.

We conducted the validation test using the following function:

```python
LINUX_EVENTS = [
    'ADD_USER', 'DEL_USER', 'MODIFY_USER', 'ADD_GROUP', 'DEL_GROUP',
    'MODIFY_GROUP', 'DEL_LOG', 'MODIFY_CONFIG', 'FAILED_SUDO', 'CHGRP',
    'CHOWN', 'CHMOD'
]

def correctness_test(loop):
    logger.info("Starting correctness test.")

    start_count0 = coll_timeline[0].count_documents({})
    start_count1 = coll_timeline[1].count_documents({})
    logger.info(f"start_count0: {start_count0}, start_count1: {start_count1}")

    for event_type in LINUX_EVENTS:
        command = [
            'python3', '/home/bitnami/casw/generator/linux/generate_logs_linux.py',
            '-C', 'linux_test', '-L', str(loop), event_type, '-AUID', 'dummyUser'
        ]
        subprocess.run(command, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

    total_loop = loop*len(LINUX_EVENTS)
    end_count0 = coll_timeline[0].count_documents({})
    end_count1 = coll_timeline[1].count_documents({})

    while end_count1 - start_count1 < total_loop or end_count0 - start_count0 < total_loop:
        end_count0 = coll_timeline[0].count_documents({})
        end_count1 = coll_timeline[1].count_documents({})
    logger.info(f"end_count0: {end_count0}, end_count1: {end_count1}")

    riskscoreIsSame = check_riskscore()
    timelineIsSame = check_timeline()
    return riskscoreIsSame, timelineIsSame
```

The `correctness_test` function takes an argument, loop, and generates all the events in `LINUX_EVENTS` for the specified number of iterations. After updating the collections for the given loop iterations, it calls `check_riskscore` and `check_timeline` to verify whether the data in both collections are identical for `BNWEngineV4` and `BNWEngineV2`.

Some setup is required before running this function. Each `BNWEngine` updates different collections in the database, and these collections are cleared of data prior to each test. We ran the test multiple times with `loop = 10000`, and the results consistently showed that the output of `BNWEngineV4` matches that of `BNWEngineV2`.

2.2 Improvement of Multiprocessing

In this section, we highlight the improvement in processing speed achieved by using multiprocessing in `BNWEngine`.

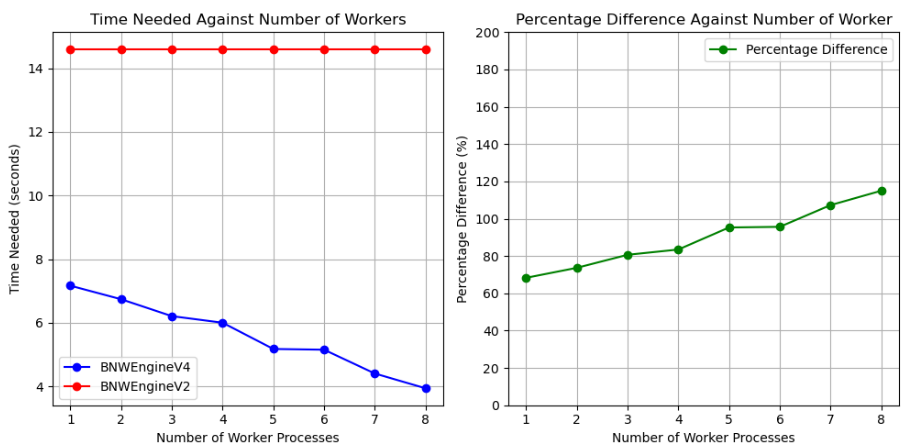

Figure 1: Sending Alerts via Route A and Increasing Worker Processes

Figure 2.2 presents the results of an experiment where 100 alerts were sent to both `BNWEngineV4` and `BNWEngineV2`. It is important to note that `BNWEngineV2` does not support multiprocessing, so increasing the number of worker processes does not enhance its performance.

In this test, both engines calculate the risk scores for entities without any historical alert data. To ensure this, the alerts are sent via Route A to prevent them from being cached. This setup minimizes any performance improvements that could result from better cache design, which will be discussed in Section 2.3.

As shown in the results, increasing the number of worker processes leads to a significant reduction in the time `BNWEngineV4` requires to process the 100 alerts, with improvements ranging from approximately 60% to 120%.

2.3 New Cache Design

In the tests described in this section, varying amounts of alert data are populated into the alerts collection in the database for each test. Additionally, the number of worker processes used by `BNWEngineV4` is limited to one, and alerts are sent via Route B.

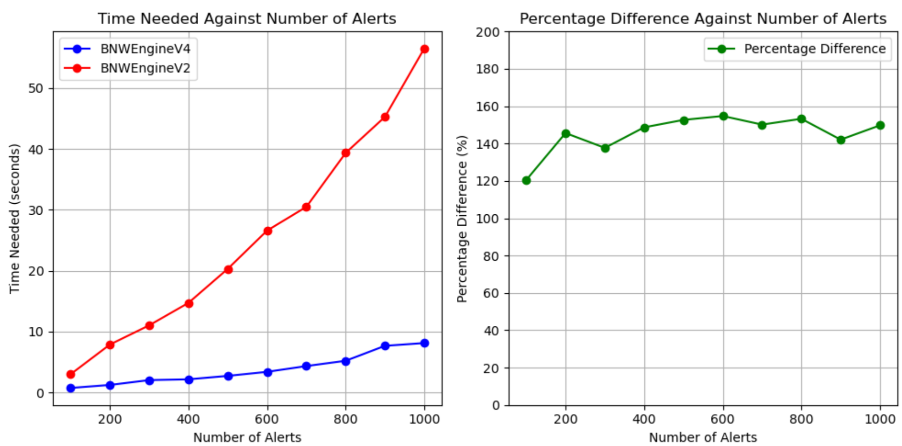

Figure 2: Sending Alerts via Route B

As shown in Figure 2.3, as the number of alerts increases, the difference in processing time between `BNWEngineV4` and `BNWEngineV2` becomes more pronounced. This is because more data needs to be retrieved and stored in the cache of `BNWEngine`. `BNWEngineV2`, with its inefficient cache design, requires significantly more processing time as the number of alerts increases.

In conclusion, the new cache design improves the performance of `BNWEngine` by approximately 120% to 160%.

2.4 Optimized Code

In this section, the number of worker processes used by `BNWEngineV4` is limited to one. Additionally, to minimize the performance improvement attributed to a better cache design, the alerts will be sent via Route A.

As shown in Figure 2.4, as the number of alerts sent increases, the processing time required by both `BNWEngine` versions also increases. However, `BNWEngineV2` requires significantly more processing time than `BNWEngineV4`. Specifically, `BNWEngineV2` takes approximately 50% to 70% longer to process the alerts compared to `BNWEngineV4`.

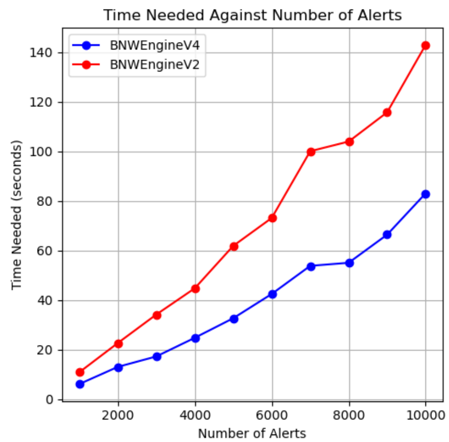

Figure 3: Sending Alerts via Route A and Increasing Alerts

2.5 Performance Difference Under Normal Circumstances

In this section, we will simulate real world scenario as close as possible. For instance, we will populate the database with 10000 alerts before running the tests. `BNWEngineV4` will create 8 worker processes as there are 8 logical cores in our machine. Then, we will send the alerts via Route A so that the alerts will be updated to database and cached in `BNWEngine`.

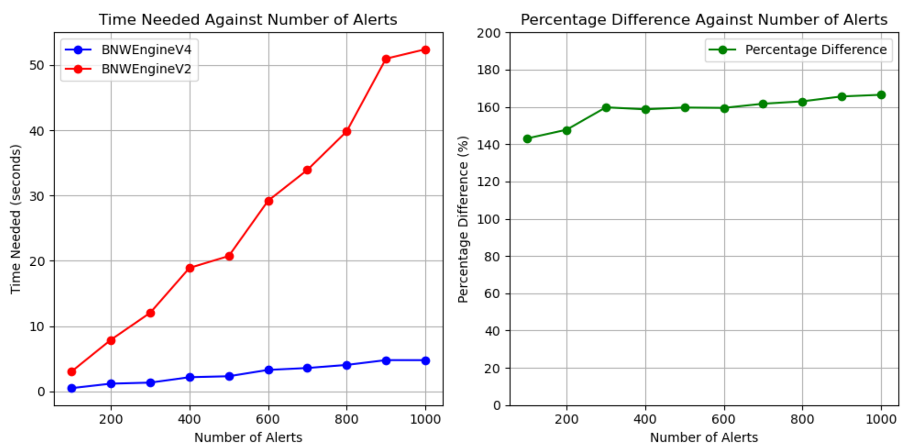

Figure 4: Simulation of Real World Scenario

From Figure 2.5, we observe that the performance improvement of `BNWEngineV4` ranges from approximately 140% to 170%. We can confidently conclude that `BNWEngineV4` is indeed faster than `BNWEngineV2` under all circumstances, based on the findings in Section 2.