Android Evidence Database:

For Forensic Use

Executive Summary:

With technology becoming more and more integrated into the everyday life, digital forensics has begun to play a larger role in proving the innocence or guilt of suspects. One piece of

technology that is a staple nowadays are cell-phones. Mobile apps, which are located on the phones create records of all sorts of digital evidence such as GPS locations, activity timestamps, visited URLS, web history, social media contacts, etc., that is either saved on the device or on their own servers. Current digital forensic practices often involve manually combing through all the files, shared-preferences, and databases on the mobile device. This process is time consuming and error prone.

Our task is to create a real-world evidence database of Android applications from many app stores that are globally used. We will create web crawlers to traverse the app stores collecting metadata and downloading the application. After collecting the application file, we will run it through the forensic analysis tools to collect where the application is storing the information that it gathers. Having this information stored in the database, we will then allow users to request the information about a specific application.

Requirements Specifications:

Functional Requirements:

- For all apps on an app store.
  - APK crawler must collect APK files. In addition to collecting the metadata, the crawler will also download their current and past APK files.
  - APK crawler must store data in a database. Once the app metadata and its APK are collected, it will store all the information into the database and the APK file into the filesystem.
  - App storage data must be retrieved by forensic program. Once collected, the APK will be passed through our client's forensic program that will output additional information about where the app stores data on a user's device.
- Database must store results from forensic program. Once the data has been passed to the forensic program it must store the results outputted.
- Backend
  - Process all requests from the frontend
  - Allow new forensic reports to be uploaded
- Website / UI
  - Query database for specific application
  - Query database given a list of applications in CSV format

- Display application metadata and report data

Users and use cases:

The primary intended end user for our database will be digital forensic investigators. There are several different scenarios in which our database will be useful to a digital investigator. One such situation might be in utilizing the location data gathered from an application, associated with the timestamps, to prove that a client was not in a certain place at a certain time. Investigators and their teams will be able to go to our web interface and search for all the applications which are present on the mobile device. Our software will tell them which applications contain the desired metadata.

Another possible user of our database will be academic researchers who study mobile applications. As this database will be the largest collection of Android APK files to date, researchers studying anything related to mobile applications and their contents may find use in the ability to search through all the available applications across all the third party app stores.

Non-Functional Requirements:

- Scalability - The system must be scalable to support all the vast amount of applications we need to download and analyze. This will be done by designing our system using microservices and using a NoSQL database.
- Availability - The system must be operating 24/7 to ensure that all versions are collected when they are updated on an app store. In addition, the service needs to be available 24/7 to support 3rd party requests at any time.
- Maintainability - The system needs to be maintainable past the day that we deliver the product. Since webpages are constantly updated and changed the crawlers also need to be updated to support those changes. This will be done by creating detailed documentation on the product and designs of how each component should behave.
- Data Integrity - The data that we collect from the websites and the forensic tool should not be modifiable by any individual after collection. We will ensure that the data remains genuine by restricting access to the database and filesystem. We will also replicate our database to ensure that if there is a database failure we will not lose any data.

System Design & Development:

Design Plan:

In regards to our crawler design, we decided to design each crawler with a unique skeleton for visiting each application on a given store. In order to allow for ease of creating new

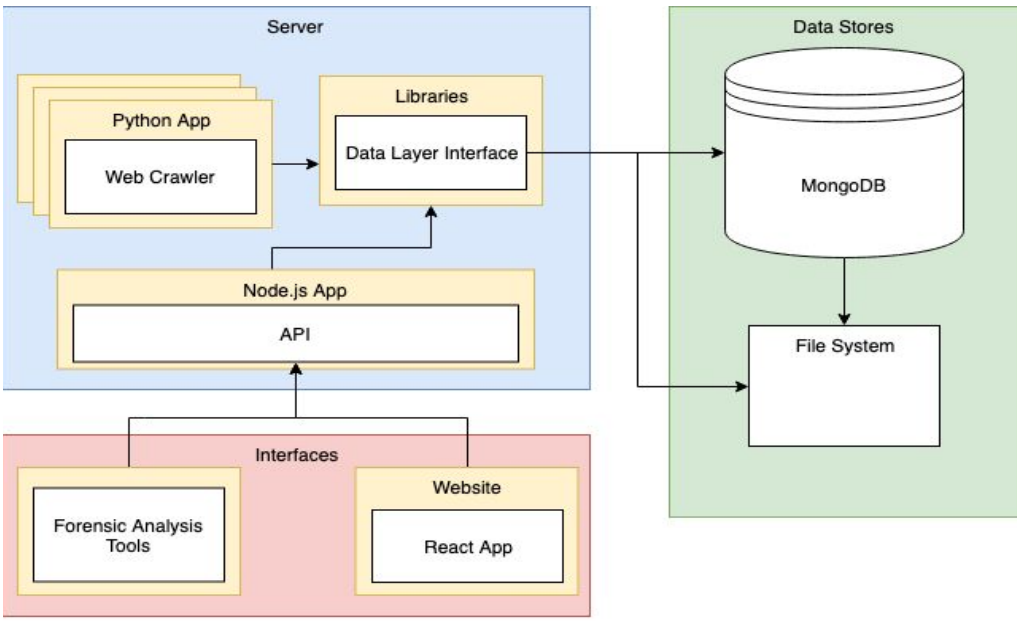
crawlers, it was decided to extract the common components. This would allow for each crawler to only have to implement the app location and metadata retrieval on the site.

For our design we decided to use python scripts for the web crawlers, MongoDB for our database and use the LSS drive as our filesystem for storing the APK files. Each crawler will be contained as a separate service, allowing us to modify and scale the crawlers individually based on their needs. We decided to use MongoDB as our database because we wanted a database that could scale to meet our needs. MongoDB can scale horizontally which allows us to grow fast as more data is collected. We decided to go with a filesystem to store the APK files as they will not be accessed often and there will potentially be large files that would be difficult to store in a traditional database. Figure 1 below shows what our designed system will look like.

Design Objective, Constraints, Trade-offs:

Our design objective was to create a system that could collect information from websites and then store it to later be accessed by an end user. We had to take into account several considerations when we were designing the system. The first was the amount of data that we were going to be collecting. We were required to collect all the metadata about each application from each store as well as the apk file itself. We needed to ensure that will all the apps on all the stores that we would create a system that could scale and be able to handle the influx of data that we would be collecting. We also needed to make sure that our system was easily extensible. With so many app stores available throughout the world, we needed to make sure that our system would allow for new crawlers to be added to system easily.

Architectural Diagram, Design Block Diagram:



(Figure 1. Architectural Diagram)

Our system is constructed with four distinct components. Theses components are the frontend, backend, crawlers, and data stores. The next section will go into more depth about each component.

Description of Modules, Constraints, and Interfaces:

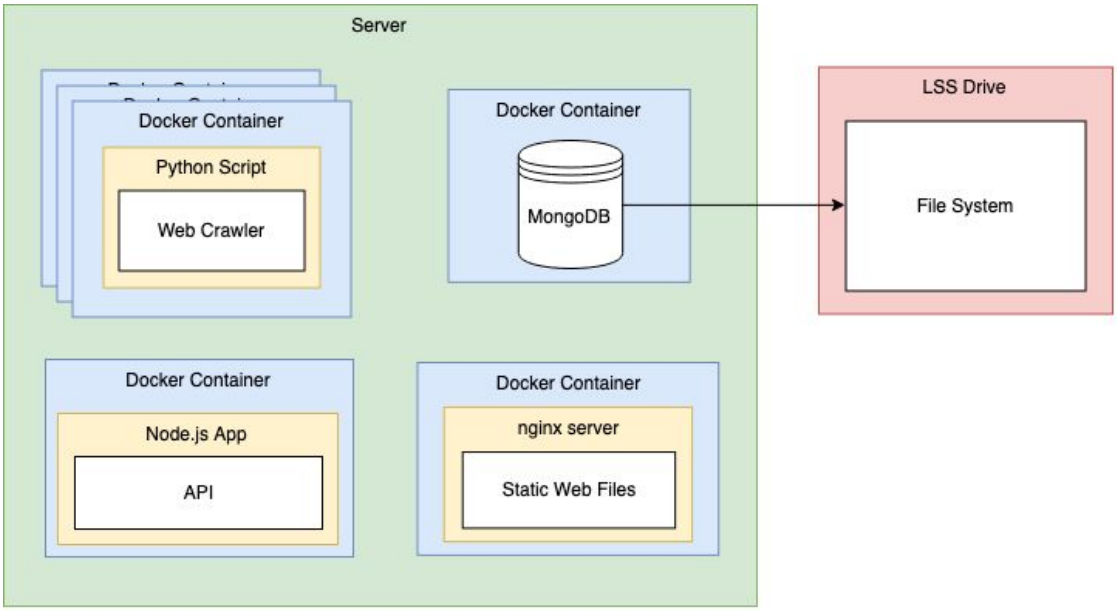
Frontend - The frontend of our system is a website. The website allows end users to query our database and extract the information that they require.

Backend - The backend application is responsible for acting as an intermediary between the frontend and the database. The backend is also responsible for allowing users to upload new forensic reports for a particular application version.

Web Crawler - The web crawlers are used to scrape web pages for information. The crawlers visit a page about a particular application extracts all the information about that application, downloads the apk and then saves it to the database and filesystem.

Data Stores - The last main component of our system is the data stores that we use to store all the information we collect. We store all the metadata that we collect for an application in a database and the actual apk files in a filesystem.

Implementation:



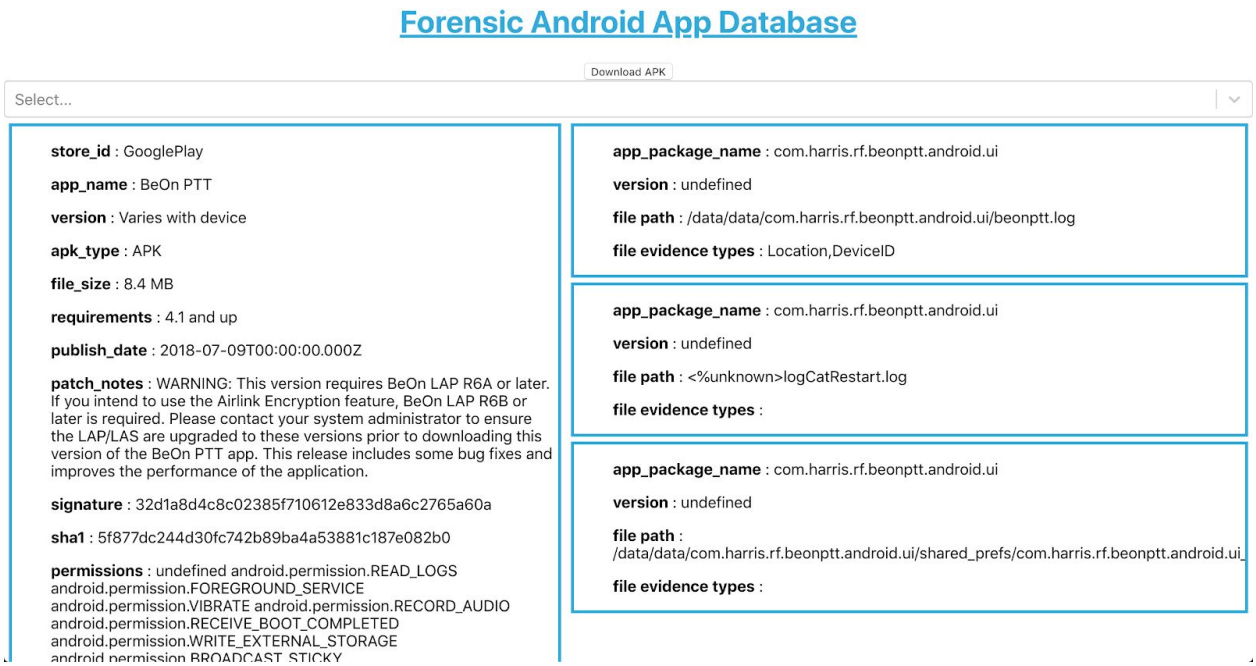
(Figure 2. Implementation Diagram)

Figure 2 represents how we have implemented our solution. Each service is a separate docker container deployed on the server. All of our data is stored on the LSS drive hosted at Iowa State University.

Implementation Diagram, Technologies Used

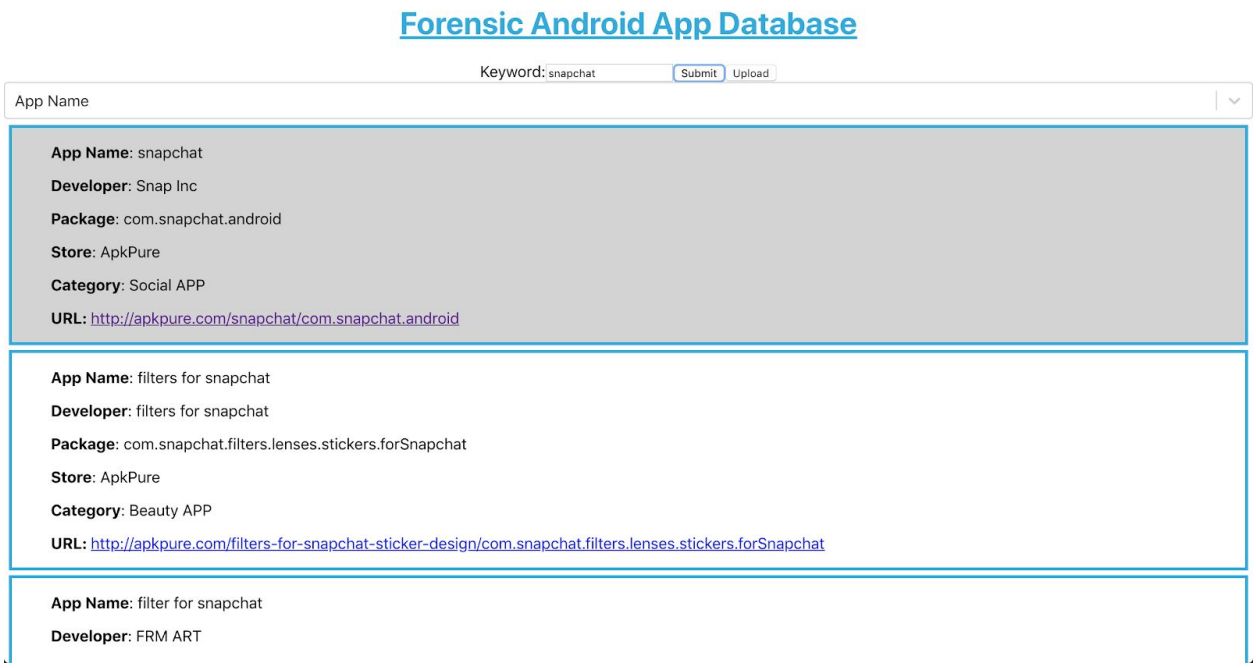
Choices:

Frontend - For the frontend we chose to use React as our framework. We use Nginx to serve our static web files generated from our build. We use a CSV formatted file to allow users to do large queries with the database.



(Figure 3. Application View on Website)

In Figure 3, we see how the interface appears after selecting an application. In this instance, we have selected the application “BeOn PTT” and are given a handful of metadata related to it. On the right side of the screen, report data will be displayed if there is data in the database.



(Figure 4. Search Results on Website)

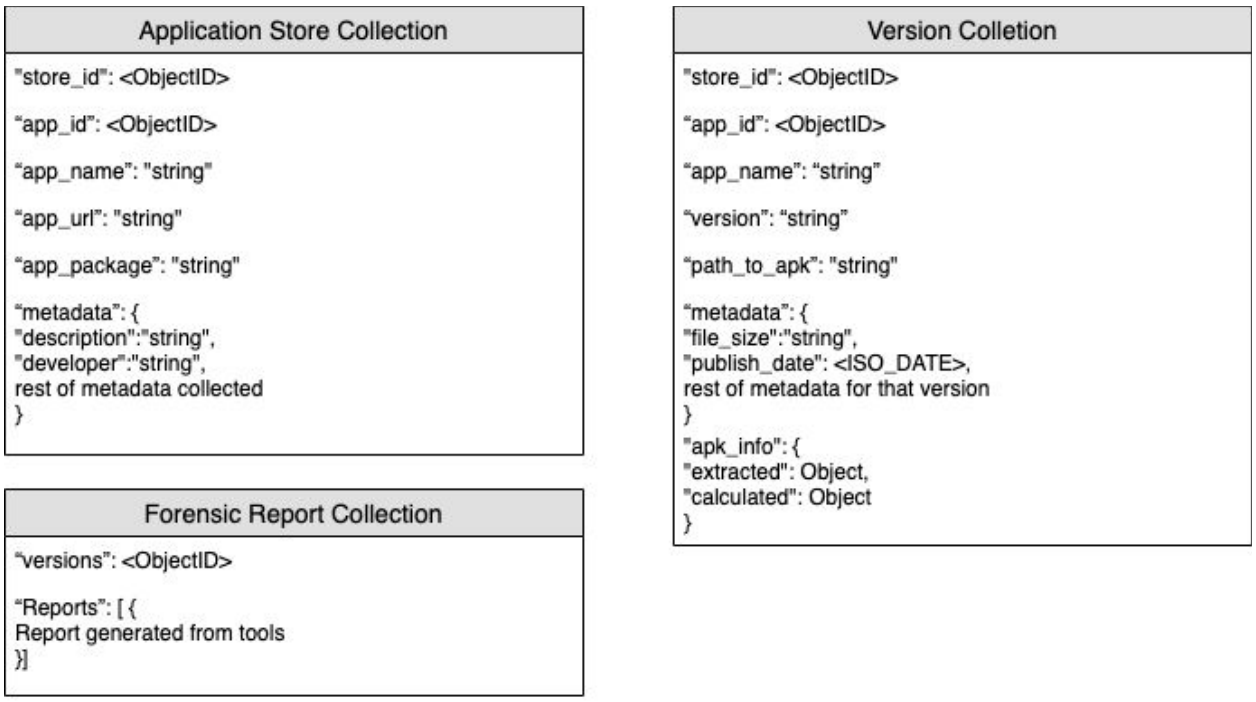
Figure 4 shows us the UI when a user searches for the application “snapchat”. The UI then displays a list of application that are closely associated with that keyword. Each app also has some metadata displayed with it.

Backend - The backend is a Node.js application. It utilizes the Express.js framework for routing and Mongoose to communicate with the database. We choose to go with Node to make it a full javascript stack and it is a simple tool to create an API with. Node also has a solid library to communicate with our MongoDB. Below is the API that we have created to interact with our system:

- Returns a list of applications matching the query parameters of app name or package name
    - GET /api/v1/applications?appName=”string”&packageName=”string”
- Returns detailed all detailed information about a specific application
    - GET /api/v1/applications/{app_id}
- Returns detailed info about a list of applications requested
    - POST api/v1/applications
- Downloads a specific apk file
    - GET api/v1/download/{version_id}
- Returns statistics about the database
    - GET api/v1/stats

Web Crawler - The web crawlers are python scripts that utilize the beautifulsoup library to scrape info off of websites. The crawlers also use selenium to create a headless browser to navigate to the web pages. In order to keep consistency between crawlers, all communication between them and the data stores have been contained in a separate script for ease of access. This script also contains functions for retrieving web pages with a built-in system to prevent rate-limiting.

Data Store - The two data stores that we choose are MongoDB and the filesystem. We choose to go with MongoDB as it meets our performance needs and also allows us to scale horizontally easily. MongoDB also gives us the flexibility to add fields based on websites as each site contains different information to collect. Below you can see our database model that we have constructed.



(Figure 5. Database Model Diagram)

The database model consists of three collections. The first collection is the application collection. The application collection contains one record for each application on each store with all the metadata related to it that is collected. The version collection contains all the info about a specific version for a specific application. The forensic report collection contains all the reports that are generated from analyzing the apk files.

We choose to use the filesystem to store our apk files as we will not be accessing them regularly and the file sizes can be quite large, which a traditional database would not be able to handle well. We are utilizing the large scale storage solution (LSS) at Iowa State University for this aspect of the solution.

For building and deploying this system we plan on using docker containers for each of our services. Each service will be its own container in order to isolate each service from one another and make them more secure. Additionally, by making them all containers we can deploy each of our services faster, update them in an automated fashion and create more instances of a crawler to crawl an application store faster.

Testing, Validation, and Evaluation:

Test Plan:

We used manual testing for the majority of our project. The only automated tests we implemented are unit tests for web crawlers.

Unit Testing:

Web crawlers have automated unit tests that run on startup. These tests use a local copy of a portion of the associated crawler's app store's site. The test html pages will be crawled through and the results will be compared to a predetermined correct data set. Currently we only have one crawler set up to accomplish this.

Success condition: 100% data match

Results: Currently the unit tests for the web crawlers have only been implemented into one of the web crawlers. This is because the client emphasized we get as many crawlers as possible working, rather than focusing time into unit testing. The crawler with this implemented matches the predetermined data set with 100% success.

The backend has automated testing to ensure that all the API endpoints behave as expected. We have tests to ensure that the API returns the correct result for both good and bad requests.

Success condition: 100% tests pass

Results: The backend passes its unit tests with 100% success.

Interface Testing:

Web Crawlers → Database:

The web crawlers have been tested to ensure that they can communicate with the database correctly. All metadata must be written to the database and each app from each app store must have its own entry.

Success condition: 100% of data is written correctly

Results: The crawlers record metadata on the database with a success rate of 100%. This is feasible due to the nature of our MongoDB database where entries do not need to be in a set structure.

APK files are checked for duplicates before download, so the database is tested for any duplicate files due to potential signature collisions.

Success condition: Less than 5% of apps have duplicates

Results: Currently our database reports a duplication percentage of 0.3 percent. This number is probably not representative of the true value as we did not have duplication metrics implemented when collection began, so numbers have only been updated with newly collected apps. In addition, since Google Play only provides one version, this reduces the possibility of having a duplicate. We expect this number to grow as more stores are implemented, especially ones where we have access to more versions of the app. However, this value should still be under 5%.

User API → Database:

The user API that forensic analysts will use to get information about specific apps was tested to ensure that it communicates with the database correctly. The API must be able to query metadata and APK files from the database.

Success condition: 100% of data is correctly obtained

Results: The website's queries all reach the API endpoints and receive accurate apps and metadata with a 100% success rate.

Database → File System:

The database was tested to ensure that all file paths point to the correct files in the file system.

Success condition: 100% of files are in the expected directories

Results: All files in the database point to the correct files in the file system. The connection between the app and the location are determined when the APK file is downloaded initially.

System Integration Testing:

Tests were run to make sure that apps and metadata follow the data path and are written into the database and stored in the file system correctly. We verified that metadata and apps were correctly obtained on the frontend website.

Validation:

With close inspection by our advisor, we have determined that we have followed the guidelines laid out for us. The project accomplishes most of the goals that were established at the beginning of the academic year, and we have prepared information for the next group to continue the project.

Project and Risk Management:

Task Decomposition:

Connor - Crawler Implementation

Emmett - System and database design. Database, backend and docker implementation

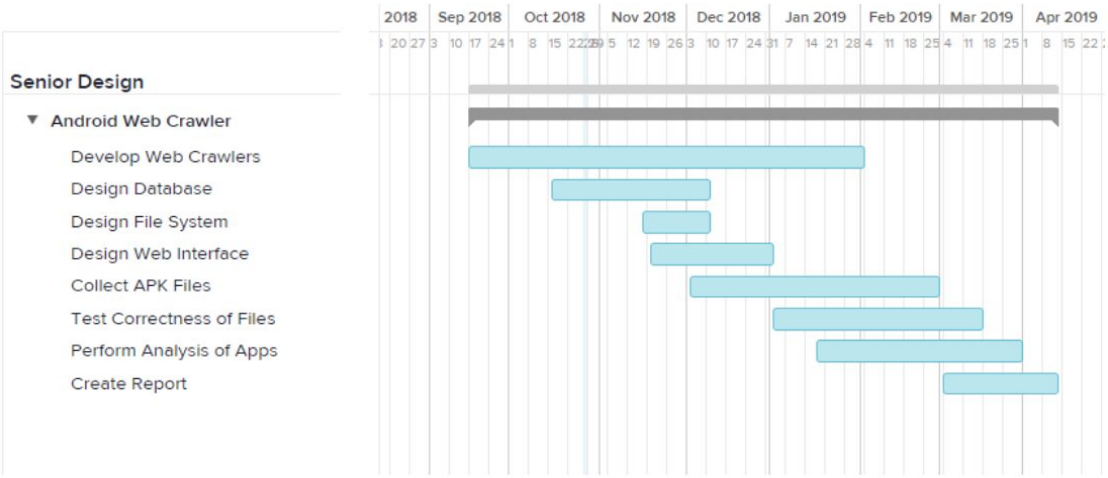
Jake - Crawler Implementation

Matt - Crawler and Frontend Implementation

Mitch - Crawler Implementation

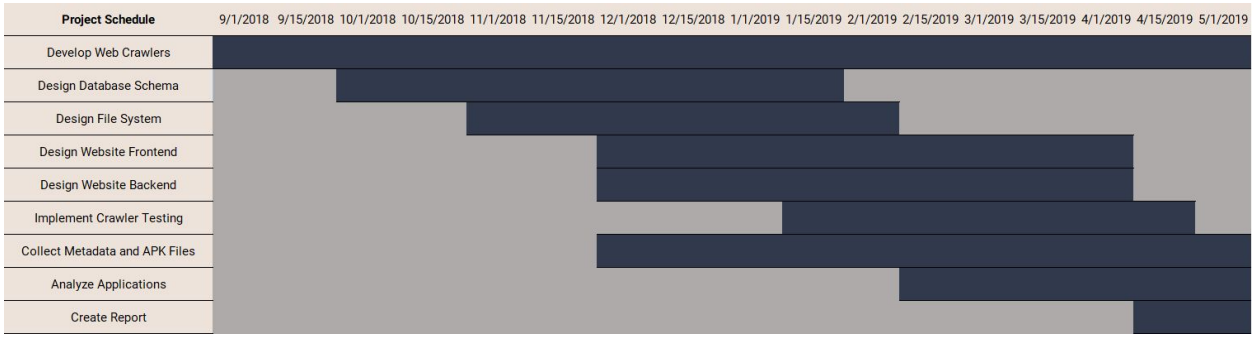
Project Schedule:

Proposed:



(Figure 6. Proposed Gantt Chart)

Actual:



(Figure 7. Actual Gantt Chart)

Figure 6 represents our anticipated progress during the first semester of our class. It was mostly based off of the schedule laid out by our clients. Figure 7 is our actual progression. We ended up developing web crawlers throughout the whole class and many of our milestones lasted longer than originally planned.

Risks and Mitigation:

Anticipated:

Our project requires that we obtain a large amount of storage space to store all the applications that we download. We might not be able to obtain all the space needed to download all the APK files. We are looking into possible solutions that we can utilize at the scale we need.

Another resource that might be difficult in obtaining at the scale we require is computing power. Our solution will be crawling multiple app stores across millions of pages requiring a significant amount of computing power be dedicated in parsing all of the web pages. Currently we are talking to CSAFE to provide us with the compute power that we need. In the meantime we can use a VM from ETG to test our service on a smaller scale.

We are implementing the solution in python and using MongoDB for our database. We as a team do not extensive knowledge or have worked on a large project in python. This will require us to ramp up our understanding of python. In addition, none of us have worked extensively with MongoDB before which is another piece of technology that we will have to learn.

Actual:

One risk that we encountered during the project was the legal issues surrounding crawling over the stores and storing the APK files. This risk was particularly important as it was the core of our project. There were concerns about whether we were violating the terms of

service of the stores by using code to collect all of the apk files and metadata. In addition, there were also concerns of storing the APK files and redistributing them as they may contain copyrighted material that we would then be in violation of. We did not deal with this risk directly but shifted the responsibility of the problem from us to our clients.

Another risk that we encountered was our lack of knowledge with several of the technologies that we were using. This included how to crawl over websites, MongoDB, Docker and other small technologies used in the project. We were able to mitigate this risk by doing research into the technologies that we wanted to use and looked up best practices for how to use and implement the technology.

A big risk for us was to ensure that we would have enough space to store the APK files that we were downloading. With the amount of apps available on the stores, we needed to make sure we had plenty of space to store them all. We were able to secure storage space that meet our standards through the large scale storage solution here at Iowa State.

Lessons Learned:

As we worked on this project we learned a lot of useful information pertaining to how it is to work for a client as well as building our product from the ground up. It was a good learning experience in planning out what we want to achieve and how we planned to go about accomplishing our set goals. We also learned many technical skills including python and how to navigate through html as well as working with large quantities of data.

Conclusion:

Closing Remarks for the project:

Our solution will be able to reduce the time taken for digital forensics. It is able to do this by utilizing the database we created. Forensic analyzers will be able to query our system by entering the name of applications of the digital device and our system will return the type and location of the information the app catalogs. This transaction of information will be much faster than combing through every file on the device.

Future Work:

There are several things that can be continued on our project. One direction is to develop crawlers to add to the amount of information that we collect as we did not get every app store available. Another direction is starting to make this project ready for use in the field. This includes refining the website to make a better user experience and add additional features. Also making our system ready for a production environment like more testing, increased security, and making our containers more resilient.

Team Information:

Connor Kocolowski: Senior in Computer Engineering

Emmett Kozlowski: Senior in Computer Engineering

Mitchell Kerr: Senior in Software Engineering

Matthew Lawlor: Senior in Software Engineering

Jacob Stair: Senior in Software Engineering