On Reducing Network Usage with Genetic Improvement

James Callan, William B. Langdon, Justyna Petke
James.Callan@cytal.co.uk,w.langdon@cs.ucl.ac.uk,j.petke@ucl.ac.uk
Department of Computer Science, University College London, Gower Street, London, UK

ABSTRACT

Mobile applications can be very network-intensive. Mobile phone users are often on limited data plans, while network infrastructure has limited capacity. There's little work on optimizing network usage of mobile applications. The most popular approach has been prefetching and caching assets. However, past work has shown that developers can improve the network usage of Android applications by making changes to Java source code. We built upon this insight and investigated the effectiveness of automated, heuristic application of software patches, a technique known as Genetic Improvement (GI), to improve network usage. Genetic improvement has already shown effective at reducing the execution time and memory usage of Android applications. We thus adapt our existing GIDroid framework with a new mutation operator and develop a new profiler to identify network-intensive methods to target. Unfortunately, our approach is unable to find improvements. We conjecture this is due to the fact source code changes affecting network might be rare in the large patch search space. We thus advocate use of more intelligent search strategies in future work.

CCS CONCEPTS

• **Software and its engineering → Search-based software engineering**.

KEYWORDS

genetic programming, genetic improvement, SBSE, HTTP, GIDroid, Robolectric

ACM Reference Format:
James Callan, William B. Langdon, Justyna Petke . 2024. On Reducing Network Usage with Genetic Improvement. In *Proceedings of 13th International Workshop on Genetic Improvement (GI @ ICSE 2023).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Constant increase in network usage of mobile devices is causing major issues, both for mobile device users and network managers. Mobile traffic on cellular networks increased by 7 fold between 2016 and 2021 [33]. This increase required massive infrastructure investment, and greatly increased energy consumption of cellular networks [5]. It is estimated that the consumption of these communications could be responsible for up to 23% of greenhouse gas emissions by 2030 [5]. More data usage can result in more expensive bills for users, and increase in unresponsive applications, waiting to fetch online assets. Khalid et al. [21] identified excessive network usage as one of the most complained about issues of Android applications.

Work improving network usage for Android applications has mostly used prefetching [9, 20, 27, 37]. However, the primary aims of prefetching are to save energy and avoid having to wait for assets to be downloaded when they are needed, improving the responsiveness of applications. If a prefetching scheme is too aggressive it may fetch assets that are never actually needed and thus increase network usage. Li et al. [23] were the first to show that network usage can be optimized by a more light-weight approach of applying modifications to source code, although their primary goal was reduction of energy consumption. Li et al. [23] proposed to simply bundle together HTTP requests into single, larger requests. Whilst this approach was successful, it is not able to produce many of the kinds of changes that we found that developers make to improve network usage [13], such as adding conditions to network requests and caching variables.

Given the different changes to source code that developers make to improve network usage of their applications, we need an approach that could automatically navigate this search space to find improvements. Therefore, we propose to use Genetic Improvement (GI) [30] for this task. GI has proven successful at improving a wide array of program properties. This includes program repair (e.g., [7, 19, 36]), execution time (e.g., [22, 29, 32]), memory usage (e.g., [8, 16, 35]) and energy usage (e.g., [10, 12, 34]). Moreover, genetic improvement has already been applied to Android applications, providing promising results. For instance, Bhokari et al. [10] improved energy usage of applications, while we have improved their responsiveness [15]. Moreover, we have achieved only limited success at improving the frame rate of Android applications [14].

More recently, we attempted to improve network usage of Android applications using multi-objective GI, concurrently targeting execution time and memory usage as improvement objectives [17]. We were able to successfully improve memory usage and execution of Android applications, however, we found we could not reduce network usage. We believe that this limitation was due to two reasons: 1) the benchmarks did not generate enough network traffic to offer the chance for improvement; and 2) some modifications to code we used, although potentially beneficial for memory and runtime improvements, could not achieve network usage reduction.

Therefore, we propose to try to overcome these limitations by: 1) developing a profiler to identify the most network-intensive areas of code; and 2) applying genetic improvement with novel, network-specific mutation operators to the identified code.

We extend our GIDroid framework [17] and apply our approach to 7 applications that we have identified as being network-intensive. Unfortunately, no improvements were found. We conjecture the

GI @ ICSE 2023, 16 April 2024, Lisbon

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

space of changes is too sparse for our approach, and thus more intelligent search strategies need to be considered in future work.

Overall, with this work, we provide the following contributions:

(1) A new profiler to identify network-intensive methods in Android applications.
(2) A set of Android applications that use network extensively, for future research in network usage optimization.
(3) A set of mutation operators that mimic modifications made by developers that improve network usage, including a novel operator that avoids unnecessary HTTP requests.
(4) An empirical study showing the effectiveness of our approach at finding improvements to network usage.

Our framework and profiler, as well as all the results obtained, are available on the following website, so others can verify and extend our work: https://github.com/SOLAR-group/NetworkGI

The rest of the paper is structured as follows: Section 2 describes how we refine the approach in the previous work to improve network usage. Section 3 lists the research questions (RQs) we have about our approach for improving network usage. Section 4 describes the methodology we use to answer our research questions. Section 5 details the results of our experiments. Section 6 explains the threats to the validity of our work and how we mitigate them. Section 7 lists the conclusions of our work.

2 APPROACH FOR IMPROVEMENT OF ANDROID APP NETWORK USAGE

We use the framework presented in Figure 1. GIDroid [17] is a framework for applying genetic improvement to Android applications. GIDroid generates variants of Android applications in the form of lists of edits to their source code. These edits include the "traditional" GI operators, which can copy, replace, delete and swap statements in the abstract syntax tree (AST), along with two caching operators to avoid unnecessary method calls. These variants are then improved with the use of search algorithms (such as genetic programming). GIDroid randomly generates variants and validates them with unit tests, particularly tests written in the simulation-based library Robolectric [4]. This saves on the time needed to transpile, package, and install the applications on actual devices or emulators, speeding up the GI process. If all tests pass, the variant is considered valid, if it is invalid its fitness will be set to the worst possible value[1]. During test execution, the non-functional property/ies being improved can be measured and then set as the fitness for valid patches. The fitness is then given to the search algorithm to generate further variants.

We modify the framework in two key ways. Firstly, rather than using the PMD static analyzer [31] to detect target methods for modification, we develop our own profiler, specifically for identifying methods that make large HTTP requests. The PMD static analyzers performance patterns only concern memory usage and execution time, so with a more appropriate profiler, we may be able to achieve better results. Secondly, we modify the set of mutation operators used to more closely reflect the changes made by software developers [13].

[1]Maximum value for Java's Float.

2.1 Network Usage Profiler

In order to identify the most network-intensive areas of code, we develop a profiler. This profiler identifies and instruments HTTP requests made in three popular HTTP libraries in Android (`HttpUrlConnection`, `OkHttp`, and `volley`). Both `volley` and `HttpUrlConnection` are official Android HTTP libraries, whereas `OkHttp` is a popular third-party library (appearing in almost 5% of all applications in the Google Play store [6]) which is in fact used as the backend of `HttpUrlConnection`. We show example requests made by each library in Figures 2, 3, and 4. We use the Soot [2] static analysis tool to find invocations of HTTP requests in each of these libraries and then exercise the application, logging the size of the data that is sent and received. In the case of `HttpUrlConnection`, the static analysis looks for the invocation of the read method on a BufferedReader which is reading the InputStream of a `HttpUrlConnection` object and logs the size of the lines that are read by the buffered reader. In the case of `volley`, all overridden onResponse and onErrorResponse methods on Response.Listener and Response.ErrorListener objects are modified to log the size of the response. Finally, for `OkHttp`, we simply log the size of the body of responses that are created with the Call.execute() method. Once we have run static analysis, we can discard those applications that do not contain any invocations of the APIs of interest.

We then use automated testing tools to find the methods that result in the largest and most frequent usages of the network. In particular, we use the Monkey testing tool [3] to randomly exercise the application being profiled and exercise as much of the code as possible. We run Monkey with 1000 random inputs. Whilst other more advanced automated testing tools are available (e.g., Mahmood et al.'s EvoDroid [25] and Mao et al.'s Sapienz [26]), we choose to use Monkey as it is compatible with the latest versions of Android unlike the testing tools available at the time of experimentation.

2.2 Novel Mutation Operator For Network Use

We introduced a new mutation operator, based on our mining study [13]. In particular, we found that many developer-made changes would add conditional branching around statements, to only make the requests over the network when they were actually necessary, see bold text in Figure 5. This mutation operator wraps statements in `if` statements, with the goal of avoiding making unnecessary requests. The conditions of the `if` statements consist of comparisons between local variables and the values listed in the right most column of Table 1. In the case that the local variable is a Java primitive, direct comparison can occur. In the case where the local variable is not a primitive, we use either one of the variable's fields which is a primitive, or one of its methods that returns a primitive for comparison. In order to select a variable for comparison, we use a random selection which is weighted based on the distance between the statement being wrapped and the closest use of the variable in the AST. The probability of a particular variable v being selected given statement s is being wrapped is shown in Equation 1.

$$P(v|s) = distance(v, s) / \sum distance(v_i, s) \quad (1)$$

We use this weighting to prefer comparisons with variables that are more relevant to the statement being wrapped. Figure 6 shows

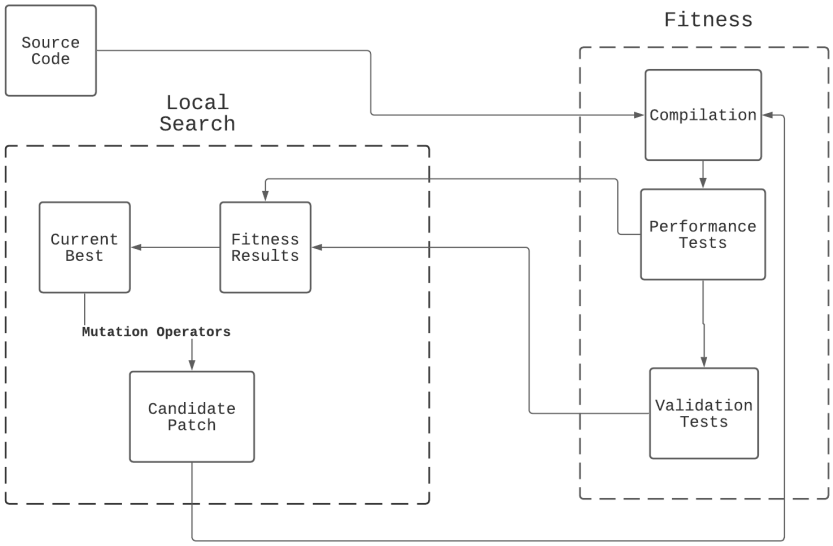

Figure 1: Overview of the GIDroid [17] framework for optimization of non-functional properties of Android applications using genetic improvement.

```
URL url = new URL("http://www.android.com/");
HttpURLConnection urlConnection =
(HttpURLConnection) url.openConnection();
BufferedReader br =
new BufferedReader(urlConnection.getInputStream());
String strCurrentLine;
while ((strCurrentLine = br.readLine()) != null) {
    Log.d("AndroidHttpProfiler", "ThisClass.getAndroid");
    Log.d("AndroidHttpProfiler", strCurrentLine.size());
    doSomething(strCurrentLine);
}
```

Figure 2: An example of an instrumented `HttpUrlConnection` request. First, a `HttpURLConnection` object is instantiated, and then its input stream is read with a buffered input stream. We instrument the code to log the method name (ThisClass.getAndroid) and the data received over the network.

how this operator is applied in practice. We select the comparison from those shown in Table 1. These conditions are based on those suggested by Brownlee et al. [11] for injecting shortcuts into code. They were extended to allow comparisons to integers from 0-5, rather than just 0, as these conditions are observed in real commits [13].

3 RESEARCH QUESTIONS

In order to evaluate the effectiveness of our proposed framework for improvement of network usage of Android applications, we pose the following research questions:

RQ1: *How much data do Android applications send over the network through* HTTP *requests*?

```
String url = "http://www.android.com/";
StringRequest stringRequest =
new StringRequest(Request.Method.GET, url,
  new Response.Listener<String>() {
    @Override
    public void onResponse(String response){
      textView.setText("Response is: "
      + response.substring(0,500));
       Log.d("AndroidHttpProfiler", "ThisClass.getAndroid");
       Log.d("AndroidHttpProfiler", response.size());
    }
  }, new Response.ErrorListener(){
    @Override
    public void onErrorResponse(VolleyError error){
      textView.setText("That didn't work");
  }
});
```

Figure 3: An example of an instrumented `volley` request. An object which extends the Request class is created and we can find the response in the overridden onResponse method.

Table 1: The potential operators and values that Java primitives can be compared with and to, depending on the type of the primitive selected for the newly created `if` statement.

Java type	Operator	Value to compare to
`boolean`	==	{`true`, `false`}
others	{==, <, ≤, >, ≥}	{0, 1, 2, 3, 4, 5}

We want to know how much of an impact HTTP requests have on network usage in Android applications and how network-intensive the methods that our profiler identifies are.

```
OkHttpClient client = new OkHttpClient();
String run(String url) throws IOException {
  Request request = new Request.Builder()
  .url(url)
  .build();
  Response resp = client.newCall(request).execute()
  String reponseString = resp.body().string();
  Log.d("AndroidHttpProfiler", "ThisClass.getAndroid");
  Log.d("AndroidHttpProfiler", reponseString.size());
  return reponseString;
  }
}
```

Figure 4: An example of an instrumented `OkHttp` request. The execute method is called on an OkHttpClient object and returns a response. As before (Figures 2 and 3), we log the method name and the data received.

(a) Code before mutation

```
...
Asset asset = assets.get(0);
Request request = new Request.Builder()
.url(asset.url)
.build();

Response response = client.newCall(request)
                .execute()
...
```

(b) Code after mutation

```
...
Asset asset = assets.get(0);
Request request = new Request.Builder()
              .url(asset.url)
              .build();
if (asset.isNeeded() == true){
  Response response = client.newCall(request)
              .execute()
}
...
```

Figure 5: An example of the 'add condition' operator, checking if the method `isNeeded` of the local variable `asset` return true. The introduced `if` statement is highlighted in bold text. This mutation avoids unnecessary HTTP requests.

RQ2: *How effective is genetic improvement at reducing the network usage of Android applications?*

We want to know if genetic improvement can automatically reduce the amount of data sent and received over the network in Android applications, and what is the impact on the amount of data sent and received.

RQ3: *How expensive is it to improve the network usage of Android applications using genetic improvement?*

We want to know how long the GI process takes to find improvements. If the process takes an exceedingly long time for small improvements it may not be worth it for developers to use GI in a real-world setting.

4 METHODOLOGY

In order to evaluate our framework for the improvement of network usage of Android applications, we propose the methodology described in this section.

4.1 Framework for Network Usage Optimization

We use our GIDroid [17], however, we modify it with the addition of our new mutation operators which specifically target network usage. Each individual program variant is represented as a patch, where each patch consists of a list of edits which is sequentially applied to the source code of the problem.

Fitness Function To evaluate each program variant i.e., whether it improves network usage without sacrificing functionality, the corresponding patch is applied to the code, and the program is run against test cases to evaluate its fitness. We instrument the applications to log the sizes of HTTP queries, allowing us to directly measure the bytes sent and received over the APIs of interest. We then use this measurement as a fitness in our search algorithms, with the goal of minimizing network usage, i.e., variants with lower fitness measurements are considered fitter.

Mutation Operators Aside from our new mutation operator (see Section 2.2), we use our 'caching' operators, and 'delete statement' operators [17], which we have shown improve bandwidth [13]. There are two types of caching operators: one caches a variable value, while another a method call. The delete operator simply deletes a randomly selected statement. The aim of each of these operators is to avoid making unnecessary HTTP requests by either removing unnecessary invocations, storing their results and reusing them, or avoiding them when some state of the application suggests that they are unnecessary.[2]

Crossover Operator Since they have shown to be successful in improving other non-functional properties, we use the same uniform crossover operator, which appends sections of individuals onto the end of others, as was used in many previous works in GI, starting with GenProg [18]. Tournament selection is used to select individuals for crossover. The first half of one is appended to the second half of the other, and vice-versa, where each edit is added with 50% probability. Each such created individual undergoes mutation.

Search Strategy We evaluate both the Local Search algorithm and the Genetic Programming algorithm available in the GIDroid framework. Genetic Programming (GP) stochastically generates a set of patches (the population) and simulates evolution upon them to find better patches. In GP each patch is applied and, if valid, its fitness (in our case network usage) is measured. The next generation is then created through tournament selection (size 2), where two individuals are randomly selected, and the fittest is added to the new generation. Mutation and crossover are then applied to add new edits and combine individuals in the new generation. This process continues for a set number of generations where a population of improved patches is produced.

[2]We opt not to use the operator proposed by Li et al. [23] as none of our benchmarks contained sequential requests.

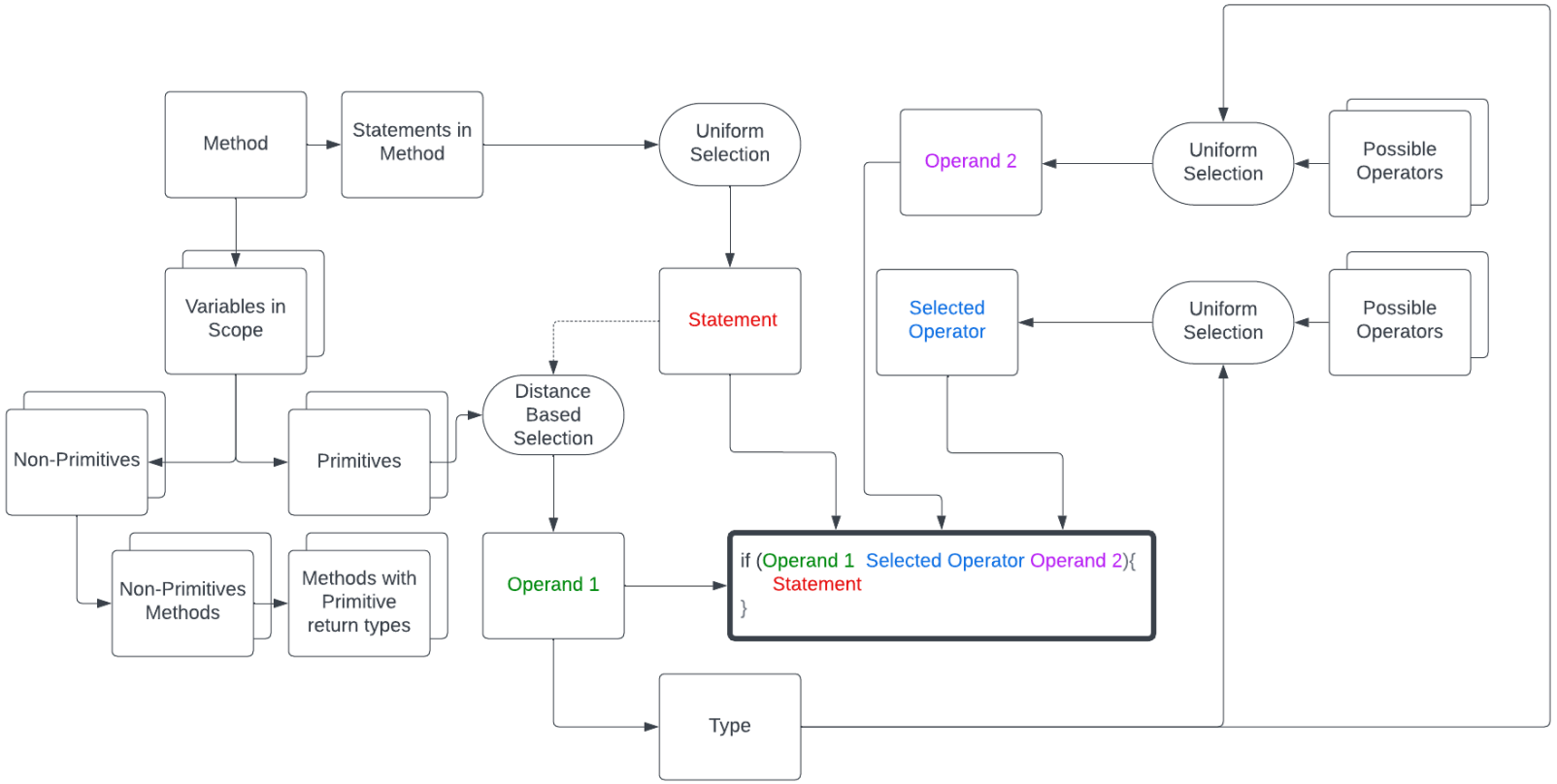

Figure 6: Process for creating a new `if` statement wrapper. First, a statement to be wrapped is selected from the target method. Next, either a primitive local variable or a method of a non-primitive local variable with a primitive return type is selected. This selection is based on the distance of the variable from the statement. Then an operator is selected based on the type of the primitive that was selected. Then, a value to compare to is selected, again, based on the type of the primitive. Finally, an `if` statement is constructed from the selected components and inserted into the target method.

In Local Search (LS), we maintain a single best individual, at each step, we add or remove and evaluate its fitness. If the new individual is fitter than the existing best it becomes the best individual. This repeats for a set number of steps where we have a single best individual. We will use 400 evaluations in each run, 400 steps in LS an 40 individuals for 10 generations in GP, as these parameters shown successful in previous work [17, 18].

4.2 Benchmark of Network-Intensive Android Applications

To evaluate our approach, we collect a set of Android applications that contain network-intensive methods. As we need tests to validate the patches that we produce, we begin by trying to find applications with network-intensive methods that are covered by local tests.

To identify these applications we run our profiler on 2 sets of applications. Firstly, we profile the applications identified by Pecorelli et al. [28] as being covered by tests in their study on the way in which every application available on the open-source app store FDroid [1] was tested. We eliminate those applications which do not contain any unit tests. Secondly, as the study by Pecorelli et al. [28] was performed in 2020, we also consider all applications that have been released since the study was performed and made available on FDroid. This resulted in a total of 4443 apps.

If our profiler identified any methods in an application that used one of the libraries previously discussed, we checked whether it also had unit tests that covered these methods. In the case where multiple methods were identified, the one that resulted in the most network traffic was selected for improvement. We use the data collected in this step to answer RQ1. In some cases, the methods identified were simply wrappers around HTTP requests, in these cases we instead ran GI on the methods which used the wrappers most often.

We also added in the F-Droid Client. Previously we [17] had failed to improve it, but now there is a possible improvement to be made based on real developer commits. Unlike the other applications being improved, this application is not the latest version but a previous version of the application, one commit before a network usage improving commit was made. We know that this improvement lies in the search space of our mutation set, thus forming a baseline for our approach. Except F-Droid Client, where we used our previous test suite [17], all benchmarks use developer-written tests.

In total, this selection process resulted in the 7 applications, shown in Table 2. Whilst we found many applications that contained HTTP API usages, the overwhelming majority were not covered by any of the application's tests. At the time of experimentation automated tools for unit test generation for Android did not support the later Android versions and, thus could not be used to generate tests for the methods of interest.

4.3 Experimental Setup

For each of our benchmarks and each search algorithm, we perform 20 runs, as Genetic Improvement is stochastic, and statistical tests are needed to evaluate its efficacy. The results of these runs will be used to answer RQs 2 and 3. We perform all of our experiments on a cloud computer with 16GB RAM and 8-core Intel Xenon CPUs.

Table 2: Android applications and commit sha checksum of version targeted for improvement and links to their repositories.

Application		Repository
Adaway	Repo	https://github.com/AdAway/AdAway
	sha	75bee423e8635f84266c521e94cf177c1521ff6c
FDroid Client	Repo	https://github.com/f-droid/fdroidclient
	sha	bf8aa30a576144524e83731a1bad20a1dab3f1bc
GPS Logger	Repo	https://github.com/mendhak/gpslogger
	sha	5437cfff42d728111f9a0ca03dc7f52a11beafc9
Mi Mangu Nu	Repo	https://github.com/raulhaag/MiMangaNu
	sha	84f8773985af04e0c96d2d5290f3f1245107c39e
Materialistic	Repo	https://github.com/hidroh/materialistic
	sha	b631d5111b7487d2328f463bd95e8507c74c3566
F-Droid Build Status	Repo	https://codeberg.org/pstorch/F-Droid_Build_Status/
	sha	818ae54b2398d1b9ec7e2ccc8f620431f001b2b6
Ooni Probe	Repo	https://github.com/ooni/probe
	sha	26dd6c96dd7129b635f15c4d4bf956939a9cdb44

Table 3: Network used by applications identified by our profiler which had network-using methods covered by unit tests, the number of KLoC in each application, and the most network-intensive method name.

Application	KLoC	Network usage (kB)	Most network-intensive method
Adaway	21.6	110.2	GitHubHostsSource.getLastUpdate
FDroid Client	88.5	237.9	FDroidApp.onCreate
GPS Logger	23.2	2.4	GoogleDriveJob.updateFileContents
Mi Mangu Nu	33.1	512.1	NineManga.getMangasFiltered
Materialistic	31.1	17.1	UserServicesClient.submit
F-Droid Build Status	7.1	1.5	FdroidClient.getRunning
Ooni Probe	32.7	147.6	MeasurementsManager.downloadReport

5 RESULTS

Next, we discuss and analyze the results which we attained from our experiments.

5.1 RQ1: Network Used

For each of the applications found in which network requests were covered by tests and thus suitable for Genetic Improvement, we measure the amount of network used by the applications with random inputs from the Monkey testing tool. The results attained are shown in Table 3.

We find that our profiler is capable of identifying methods in applications that use between 1.5 kB and 512kB of data. This data is collected over only 10000 inputs, taking a few minutes to execute. For real users, this could result in large amounts of data usage if they use the applications often, demonstrating the need for network usage reduction in Android applications.

> **Answer to RQ1:** We find that our profiler is capable of identifying methods in applications that use between 1.5 kB and 512kB when exercised with 10000 random inputs. This is only over the course of a few minutes and real usage is likely to result in large amounts of data being transmitted.

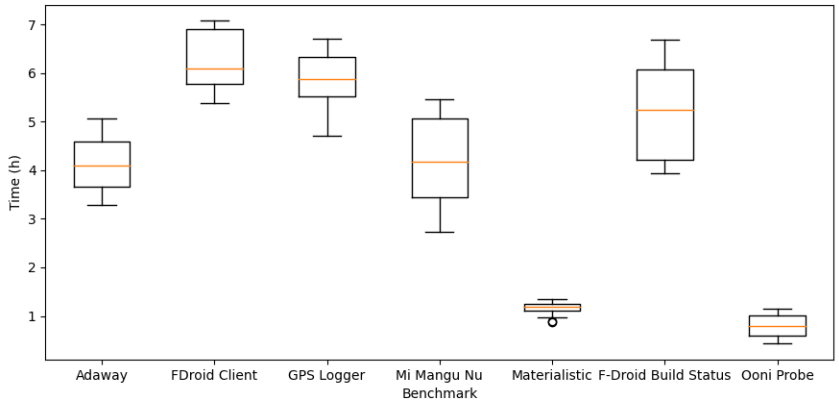

Figure 7: Time taken by Genetic Improvement when using Genetic Programming for each of our benchmarks

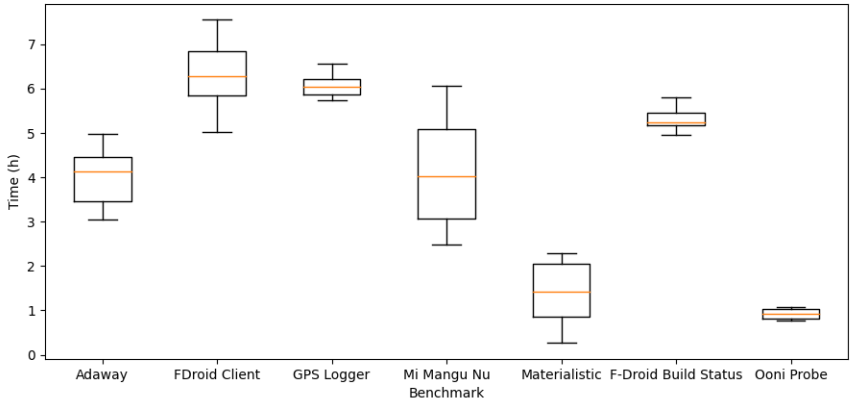

Figure 8: Time taken by Genetic Improvement when using Local Search

5.2 RQ2: Improvements to network usage

Unfortunately, we did not find any improving patches in our experiments. One possible reason is that the potentially improving mutations are too sparsely distributed in the search space. Both Genetic Programming and Local Search rely on being able to find small improvements and build upon them, so-called "exploitation". However, for this particular problem, search algorithms that can explore the search space in a more intelligent way may be more useful. Moreover, execution of a single variant on our benchmark set takes minutes to run. If we wish to perform 1000s of evaluations and explore large areas of the search space, we require faster ways to test or validate that the variant is equivalent to the original program.

To investigate this, we calculate the number of potential `if` statements that could be added to each of our benchmarks. We find that each benchmark has between 10 000 and 250 000 potential conditions that could be inserted. We show these values in Table 4. With the relatively long-running tests needed to evaluate each mutant, 400 evaluations are simply not enough to explore the 10s of thousands of potential edits in the search space in a reasonable amount of time, posing the need for effective heuristic search strategies.

Table 4: Number of potentials `if` statements which could be inserted for each benchmark (target methods in each app).

Application	Number possible `if` mutation locations
Adaway	11 088
FDroid Client	157 859
GPS Logger	32 457
Mi Mangu Nu	236 918
Materialistic	196 962
F-Droid Build Status	44 352
Ooni Probe	10 065

Answer to RQ2: We find that our approach is unable to find improvements to network usage. We believe that this is because we cannot effectively explore the very large search spaces of this problem.

5.3 RQ3: Cost of Genetic Improvement

As shown in Figure 7, running GI with the Genetic Programming Meta-Heuristic takes between 0.4 and 7.0 hours, with a median time of 4.3 hours. Alternatively, as shown in Figure 8 when using Local Search we find that Genetic Improvement takes between 0.3

and 7.6 hours, with a median time of 4.8 hours. The difference between the two approaches is due to Local Search producing more compiling variants which must then be tested, taking more time. This is not surprising as every individual in Local Search is a single edit away from a variant that compiles and passes all tests, whereas variants generated in Genetic Programming may be multiple edits away. However, this may allow GP to explore the search space more quickly than LS and be more successful in future work.

> **Answer to RQ3:** We find that running GI with the GP meta-heuristic takes between 0.4 and 7.0 hours. We find that when using Local Search GI takes between 0.3 and 7.6 hours. The median time taken by GP (4.3 hours) is less than that taken by LS (4.8 hours).

6 THREATS TO VALIDITY

Using unit tests to assess whether two programs are equivalent can lead to false positives or variants that pass all tests but are not equivalent to the original program. This threat can be mitigated through a standard code review of any patches suggested by GI by the developers of the project being improved.

Genetic Improvement is a stochastic process. This means that in some cases it can get "lucky" and find strong improvements that it wouldn't find in a normal run. We mitigate this threat by performing 20 runs for each of our benchmarks for each of our search algorithms. This gives us confidence that we know how our approach will perform in a standard run.

Whilst Madaan et al. [24] have shown that large language models can be used to improve the execution time of C++ programs, they did not consider Java Android applications. Indeed, we found we could not reproduce all the changes made by developers when simultaneously optimising multiple non-functional properties of Android applications [17].

Finally, our tool and results are available so that our work can be validated and replicated: https://github.com/SOLAR-group/NetworkGI

7 CONCLUSIONS

In conclusion, we propose an approach for the identification and improvement of network-intensive methods in Android applications. We augment an approach that was previously successful in improving execution time and memory consumption, but not network usage, to specifically target network usage by only making changes that could improve network usage. We identify 7 applications with network-intensive methods that are covered by the applications' unit tests and evaluate our approach on them. We find that our approach cannot successfully explore the tens of thousands of potential changes that could be made to our benchmarks to find patches that improve network usage. We do however provide the implementation of our tool and results so that future researchers can improve this approach and hopefully find success for this important problem.

ACKNOWLEDGMENTS

This research was supported by EPSRC grant EP/P023991/1. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising.

REFERENCES

[1] [n. d.]. F-Droid Free and Open Source Android App Repository. https://f-droid.org.
[2] [n. d.]. Soot - A framework for analyzing and transforming Java and Android applications. https://soot-oss.github.io/soot/.
[3] [n. d.]. UI/Application Exerciser Monkey : Android Developers. https://developer.android.com/studio/test/monkey.
[4] 2019. Robolectridc library. http://robolectric.org/
[5] Anders Andrae and Tomas Edler. 2015. On Global Electricity Usage of Communication Technology: Trends to 2030. *Challenges* 6 (04 2015), 117–157.
[6] AppBrain. 2023. okHttp - Android Statistics. Retrieved July 14, 2023 from https://www.appbrain.com/stats/libraries/details/okhttp/okhttp
[7] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *CEC*. IEEE, 162–168.
[8] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2018. Darwinian data structure selection. In *ESEC/SIGSOFT FSE*. ACM, 118–128.
[9] Paul Baumann and Silvia Santini. 2017. Every Byte Counts: Selective Prefetching for Mobile Applications. *Proc. ACM Interact. Mob. Wearable Ubiq. Tech.* 1, 2 (2017), 6:1–6:29.
[10] Mahmoud A. Bokhari, Bobby R. Bruce, Bradley Alexander, and Markus Wagner. 2017. Deep parameter optimisation on Android smartphones for energy minimisation: a tale of woe and a proof-of-concept. In *GECCO Comp.* 1501–1508.
[11] Alexander E. I. Brownlee, Justyna Petke, and Anna F. Rasburn. 2020. Injecting Shortcuts for Faster Running Java Code. In *CEC*. IEEE.
[12] Nathan Burles et al. 2015. Object-Oriented Genetic Improvement for Improved Energy Consumption in Google Guava. In *SSBSE (LNCS 9275)*. 255–261.
[13] James Callan, Oliver Krauss, Justyna Petke, and Federica Sarro. 2022. How do Android developers improve non-functional properties of software? *Empir. Softw. Eng.* 27, 5 (2022), 113.
[14] James Callan and Justyna Petke. 2021. Improving Android App Responsiveness Through Automated Frame Rate Reduction. In *SSBSE (LNCS 12914)*. 136–150.
[15] James Callan and Justyna Petke. 2022. Improving Responsiveness of Android Activity Navigation via Genetic Improvement. In *ICSE-Companion*. 356–357.
[16] James Callan and Justyna Petke. 2022. Multi-objective Genetic Improvement: A Case Study with EvoSuite. In *SSBSE (LNCS, Vol. 13711)*. Springer, 111–117.
[17] James Callan and Justyna Petke. 2023. Multi-Objective Improvement of Android Applications. arXiv:2308.11387 [cs.SE] arxiv, 2308.11387.
[18] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*. IEEE, 3–13. https://doi.org/10.1109/ICSE.2012.6227211
[19] Claire Le Goues, Westley Weimer, and Stephanie Forrest. 2012. Representations and operators for improving evolutionary software repair. In *GECCO*. 959–966.
[20] Brett D. Higgins et al. 2012. Informed mobile prefetching. In *MobiSys*. 155–168.
[21] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E. Hassan. .. What Do Mobile App Users Complain About? *IEEE Softw.* 32, 3 (.), 70–77.
[22] William B. Langdon et al. 2015. Improving CUDA DNA Analysis Software with Genetic Programming. In *GECCO*. ACM, 1063–1070.
[23] Ding Li, Yingjun Lyu, Jiaping Gui, and W. G. J. Halfond. 2016. Automated energy optimization of HTTP requests for mobile applications. In *ICSE*. ACM, 249–260.
[24] Aman Madaan et al. 2023. Learning Performance-Improving Code Edits. arXiv:2302.07867 [cs.SE] arxiv, 2302.07867 v1.
[25] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *SIGSOFT FSE*. ACM, 599–609.
[26] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *ISSTA*. ACM, 94–105.
[27] Prashanth Mohan, Suman Nath, and Oriana Riva. 2013. Prefetching mobile ads: can advertising systems afford it?. In *EuroSys*. ACM, 267–280.
[28] Fabiano Pecorelli et al. 2020. Testing of Mobile Applications in the Wild: A Large-Scale Empirical Study on Android Apps. In *ICPC*. ACM, 296–307.
[29] Justyna Petke et al. 2014. Using Genetic Improvement and Code Transplants to Specialise a C++ Program to a Problem Class. In *EuroGP (LNCS 8599)*. 137–149.
[30] Justyna Petke et al. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Trans. Evol. Comput.* 22, 3 (2018), 415–432.
[31] Pmd Development Team. 2023. PMD. https://pmd.github.io/.
[32] Pitchaya Sitthi-amorn et al. 2011. Genetic programming for shader simplification. *ACM Trans. Graph.* 30, 6 (2011), 152.
[33] Zsolt Temesvari and Dora Maros. 2019. Data Transfer Rates and Data Traffic Trends on Mobile Networks. *INDECS* 17 (2019), 26–39. Issue 1-A.
[34] David Robert White, Andrea Arcuri, and John A. Clark. 2011. Evolutionary Improvement of Programs. *IEEE Trans. Evol. Comput.* 15, 4 (2011), 515–538.
[35] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep Parameter Optimisation. In *GECCO*. ACM, 1375–1382.
[36] Yuan Yuan and Wolfgang Banzhaf. .. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE TSE* 46, 10 (.), 1040–1067.
[37] Yixue Zhao et al. 2018. Leveraging program analysis to reduce user-perceived latency in mobile applications. In *ICSE*. ACM, 176–186.