Energy Consumption Estimation of API-usage in Smartphone Apps via Static Analysis

Abdul Ali Bangash*, Kalvin Eng†, Qasim Jamal‡, Karim Ali§, and Abram Hindle||
Department of Computing Science, University of Alberta, Edmonton, AB, Canada *†§||
Department of Computing Science, FAST-NU, Islamabad, Pakistan ‡
Email: (*bangash, †kalvin.eng, §karim.ali, ||hindle1) @ualberta.ca, ‡i190459@nu.edu.pk

***Abstract*—Smartphone application (app) developers measure the energy consumption of their apps to ensure that they do not consume excessive energy. However, existing techniques require developers to generate and execute test cases on expensive, sophisticated hardware. To address these challenges, we propose a static-analysis approach that estimates the energy consumption of API usage in an app, eliminating the need for test case execution. To instantiate our approach, we have profiled the energy consumption of the Swift SQLite API operations. Given a Swift app, we first scan it for uses of SQLite. We then combine that information with the measured energy profile to compute E-factor, an estimate of the energy consumption of the API usage in an app. To evaluate the usability of E-factor, we have calculated the E-factor of 56 real-world iOS apps. We have also compared the E-factor of 16 versions and 11 methods from 3 of those apps to their hardware-based energy measurements. Our findings show that E-factor positively correlates with the hardware-based energy measurements, indicating that E-factor is a practical estimate to compare the energy consumption difference in API usage across different versions of an app. Developers may also use E-factor to identify excessive energy-consuming methods in their apps and focus on optimizing them. Our approach is most useful in an Integrated Development Environment (IDE) or Continuous Integration (CI) pipeline, where developers receive energy consumption insights within milliseconds of making a code modification.**

***Index Terms*—Mobile Application, Static Analysis, Energy Estimation**

I. Introduction

Smartphone apps that consume too much energy usually garner poor user reviews, leading to lost revenue [1], [2]. To ensure that an app is energy efficient, app developers have several energy profiling techniques to choose from, such as hardware-based measurement tools [3]–[5] and software-based estimation models [6]. To get accurate energy measurements using the hardware-based tools, developers must run a large number of test cases, which takes time and effort. Additionally, setting up the necessary measurement infrastructure requires developers to have specialized knowledge on smartphones and energy measurement devices, which they see as an unwanted task [7]. Furthermore, acquiring the necessary hardware can be costly, adding a financial burden. Alternatively, developers may use software-based techniques that rely on estimation [6], [8] and prediction [9], [10] models. While these techniques may not be as precise as their hardware-based counterparts, developers still prefer them because they do not require expensive equipment or specialized expertise to set them up [9]. However, developers still need access to a smartphone, as well as generate and execute test cases to measure the energy consumption of their apps.

While hardware-based techniques execute test cases to measure energy consumption, software-based techniques executes test cases to gather runtime information for energy estimation. Some runtime information, such as execution time of a task or the response time of a component, may be obtained without test case execution using static-analysis techniques, such as worst-case execution-time (WCET) [11] and worst-case response-time (WCRT) [12]. However, this information alone is insufficient to understand the energy consumption of a task [13], because software-based techniques still require additional information such as operating system logs, type of components used, and type of operations performed during execution. Moreover, creating effective test cases is a complex and costly process because it requires a thorough understanding of the app and its requirements, as well as the ability to anticipate and design for potential edge cases [14]. Running these test cases also requires significant resources such as computing time, power, and storage, adding to the overall testing expense [15]. As the software evolves, developers also need additional or updated test cases [16], [17] and such test cases might not be available all of the time.

To address the limitations of existing energy measurement techniques, we propose a static-analysis based approach that estimates the energy consumption, E-factor, of an API in a given app. We focus on API usage because API events consume 85% of the energy in an app compared to developer-written instructions (e.g., branch or arithmetic instructions) and system events (e.g., garbage collection or process switching) [18]. Instead of depending on app runtime information, our approach uses API energy profiles. An API energy profile is the base energy cost (in joules) of the tasks that the API can perform. Obtaining a single API energy profile is more cost-effective than obtaining runtime information for each individual app, because that profile may be used for all apps that utilize that API. To implement our approach, we have developed an E-factor calculator (EC) that generates the call graph of a given app and scans that app for API uses. Based on the API uses and its energy profile, EC then calculates E-factor without executing the app.

To evaluate our approach, we have studied 56 real-world apps that use the SQLite API. We focus on SQLite in this

```
1 class ViewController: UIViewController {
2   override func onTap() {
3       DB.insert(id: i, name: "Cat")
4       DB.read(id: i)
5   }
6 }
```

(a) Version-1 of the app.

```
7 class ViewController: UIViewController {
8   override func onTap() {
9     DB.insert(id: i, name: "Cat")
10    DB.update(id: i, name: "SCat")
11  }
12 }
```

(b) Version-2 of the app.

```
13 class ViewController: UIViewController {
14   override func onTap() {
15     DB.update(id: i, name: "SCat2")
16     DB.read(id: i)
17   }
18 }
```

(c) Version-3 of the app.

```
19 // SQLite API usage
20 import SQLite3
21 class DB {
22   // Updates username by id in database
23   func update(id:Int, name:String) { ... }
24   // Inserts username by id
25   func insert(id:Int, name:String) { ... }
26   // Returns all users from database
27   func read(id:Int) -> [User] { ... }
28 }
```

(d) A common class across all versions.

Fig. 1: Three versions of a sample Swift iOS app, each using the `DB` class to perform various SQLite operations.

study, because it uses the CPU, memory, and storage components of a smartphone, which are known to be one of the most power-heavy smartphone components [19]. To obtain the energy profile of SQLite, we first created a set of 24 micro-benchmarks that represent the `insert`, `update`, and `select` operations. These micro-benchmarks are freely available [20] as a useful resource for future researchers who wish to measure the impact of SQLite operations on other performance aspects, such as execution time and storage size etc. Among the real-world apps, we have compared the E-factor of 16 versions and 11 methods to their hardware-based energy measurements. Our findings show that E-factor is a reliable estimate to compare the energy consumption between versions of an app as well as between methods of an app.

II. Motivating Example

Figure 1 shows a development scenario that we will use to illustrate the limitations of current approaches for measuring and estimating energy consumption. We will then present our solution to address these limitations.

In the example, we assume that a developer produces three versions of their app, where each one uses the SQLite differently. Figure 1d shows `DB`, a helper class that implements the SQLite API, which is commonly used by all versions. In this class, `DB.read(..)` executes the `"SELECT ..."` query, `DB.insert(..)` executes the `"INSERT ..."` query, and `DB.update(..)` executes the `"UPDATE ..."` query on a database. Figure 1a shows that `DB.insert(..)` and `DB.read(..)` are called in Version-1 in a UI event for tapping the screen (Line 3). Figure 1b shows that the event handler in Version-2 calls `DB.update(..)` (Line 10) instead of `DB.read(..)`. Figure 1c shows that Version-3 calls both `DB.update(..)` (Line 15) and `DB.read(..)` (Line 16).

Depending on the task that an API performs, an API may consume different amounts of energy [18]. In our example, the SQLite API may perform *create, read, update, or delete* (CRUD) operations on a locally stored SQLite database. Each CRUD operation has a different workload and requires a different time duration to complete. Therefore, each version in Figure 1 consumes a different amount of energy. To inform developers how their code modifications in each version would affect the energy consumption of their app, we first demonstrate the usage of current energy measurement approaches, and then we demonstrate the usage of our proposed approach.

1) Using A Traditional Approach: To collect the hardware-based energy measurements, we have to first generate and execute test cases. For each version, we have written 5 test cases. In each test case, the method `onTap()` is executed 10^0–10^5 times. Following standard practice [4], we have executed each version 10 times on iGreenMiner, a state-of-the-art hardware-based energy measurement framework for iOS [21]. Our results show that for 10^0–10^5 executions of `onTap()`, the average energy consumption of Version-1 is **25.12 joules**, Version-2 is **46.45 joules**, and Version-3 is **25.45 joules**. The average time for executing the test cases per version is **121.33 seconds**.

2) Using the proposed approach to calculate E-factor: This approach does not need to execute any test cases. It instead uses an energy profile of an API that represents the base energy cost of its tasks. The energy profile of an API is obtained by measuring the energy consumption of the API's tasks on a hardware-based energy measurement setup. This energy profile is portable and can be used across multiple apps. Using the energy profile, we calculate E-factor for each version. The results show that executing `onTap()` for 10^0–10^5 reports an average E-factor value of **50.25 joules** for Version-1, **92.91 joules** for Version-2, and **50.89 joules** for Version-3.

The traditional approach typically reports a lower energy measurement compared to the E-factor estimate because an executing app goes through several architectural level performance optimizations during its runtime. These optimizations result in a significant absolute difference in the energy measurement of the traditional approach and the energy estimates of E-factor. Nonetheless, since the relative difference between the energy measurements and E-factor is consistent, we can identify the energy extensive version in our example. In this example, both E-factor and the traditional approach identify Version-2 as the most energy extensive version and Version-1 as the least energy extensive. The average time for calculating E-factor per version is **12 milliseconds**, which is **10^6 times** faster than the traditional approach.

In a real-world scenario, it is impractical to follow the traditional approach and execute all test cases at every code modification [14], especially in an IDE where a developer is continuously making changes to their code. E-factor is more suitable for this scenario because it provides energy estimates that identify if a code change consumes more/less energy, at app level and method level, in a matter of milliseconds.

III. How Do We Compute E-factor?

In this section, we first show how we obtain the energy profile of an API, which is necessary for computing E-factor. We then present our generalized approach to calculate the E-factor of the uses of an API. Finally, we instantiate our approach to calculate E-factor of an example code that uses the SQLite API.

A. Energy Profile Collection

To collect the energy profile of an API, we first have to create a set of micro-benchmarks, where each benchmark executes a separate API task. This structure enables us to measure the energy consumption of each task in isolation. We then generate the API energy profile by executing these micro-benchmarks and measuring their energy consumption on a hardware-based infrastructure. We only need to perform this process once, and then reuse its results in all future calculations. We now demonstrate the process through calculating the energy profile of the SQLite API.

1) Micro-Benchmarks Creation: We first create a simple iOS app that connects to an SQLite database stored on a smartphone. We then instrument this iOS app to create 8 micro-benchmarks for each SQLite operation (i.e., insert, update, and read), where each micro-benchmark performs an SQLite operation 10^0–10^7 times. We limit the iterations to 10^7, because executing more iterations becomes impractical as the execution thread on the smartphone tends to time out and detach from the energy measurement infrastructure.

2) Energy Measurement: To measure the energy consumption of the generated micro-benchmarks, we use iGreenMiner [21], which supports iOS apps that can run on iPhone 11 and iOS 13.4.1. iGreenMiner is a hardware-based energy measurement framework, it uses the Monsoon power monitor [22] to provide accurate energy measurements of the iOS apps. iGreenMiner can only execute Apple-script based test cases, which creates an overhead for our energy measurements. To overcome this hurdle, we extended iGreenMiner to support executing command-line-based test cases using Xcode command-line tools [23]. We have also optimized the iGreenMiner framework to decrease energy readings overhead during measurement and increase its automation capability. Since iGreenMiner requires the user to manually input the test case execution time, we have written a parser for Xcode's build log files where we store the start and tear-down times of a micro-benchmark execution. This information helps us synchronize the iGreenMiner's reported energy values with test execution time to get the exact energy measurement of an SQLite operation. In Section IV, we use this setup to measure the energy consumption of the real-world apps' versions and methods.

To calculate the energy consumption of the micro-benchmarks, we execute them 10 times each in a random order on iGreenMiner. Excluding the app setup and initialization time, this step takes an average of 1.1 $\pm$ 0.13 seconds to 2.26 $\pm$ 1.52 hours depending on the number of times that an SQLite operation executes in a benchmark. This long execution time makes it infeasible to pre-populate a database with a random number of records before each test case execution. Therefore, we keep the initial database empty for the `insert` micro-benchmarks. For the `select` and `update` micro-benchmarks, we pre-populate the database with 10^7 records to read and update from.

Table I presents the energy cost of each SQLite operation (in joules) when it executes once ($\text{Base}_{\text{exec}}$) and when it executes 10^1–10^7 times in a loop. The data shows that, after 10^2 iterations, `select` always consumes the least energy, while `insert` and `update` consume more, but similar, amounts of energy. This result shows that one API operation, in comparison to another API operation, consumes different amounts of energy even for the same number of iterations. Therefore, profiling API operations separately is important because knowing the operation execution frequency without knowing the type of operation is not sufficient to reason about its energy consumption.

B. General Approach For Calculating E-factor

There are five steps that a researcher or developer should follow to determine the E-factor of an API. To automate these steps, we have developed an E-factor calculator (EC) that works as follows:

1) *Platform selection*: select a smartphone app development platform such as Android or iOS.
2) *API selection*: based on the selected platform, select an API to measure the energy consumption of its uses.
3) *Call-graph (CG) generation*: extract the app call-graph.
4) *API call methods identification*: using CG information, identify methods that use the selected API; produce *Method-to-API* mappings by tagging these method calls with source level information.
5) *E-factor calculation*: use the Method-to-API mappings and the API energy profile to calculate E-factor.

C. Specific Instantiation for SQLite

To calculate the E-factor for the SQLite API, we discuss how we instantiate each step in our methodology:

1) Platform Selection: from the top three smartphone platforms (i.e., Android, iOS, and Windows), we chose iOS because developers prefer it for generating more revenue [24],

TABLE I: The energy profile of SQLite (in joules). $\text{Base}_{\text{exec}}$ is the cost of an operation executed once.

		# of Executions						
Operation	$\text{Base}_{\text{exec}}$	10^1	10^2	10^3	10^4	10^5	10^6	10^7
`insert`	0.00249	1.13	1.12	2.49	22.25	248.72	2,462.33	23,313.01
`update`	0.00251	1.20	1.23	2.66	25.13	249.33	2,494.38	23,677.88
`select`	0.00023	1.61	1.73	1.78	2.28	15.53	184.99	1,877.99

[25]. Furthermore, iOS development is expensive compared to the other app development platforms [26]. Since E-factor relieves the developer from owning hardware to execute test cases, E-factor becomes a more desirable choice for iOS developers.

2) API Selection: we focus on the SQLite API [27] that enables a developer to use a self-contained transactional database for persistent storage, a feature that most apps need [28].

SQLite supports several smartphone operating systems [29]. Both Apple and Google use it for their native apps [30]–[32]. Social-networking apps (e.g., Facebook), browser apps (e.g., Firefox), communication apps (e.g., Skype), and cloud storage apps (e.g., Dropbox) also use SQLite for data storage [31]. Therefore, our focus on studying the energy consumption of SQLite will affect millions of users and developers worldwide. Knowing how the SQLite API consumes energy, developers can identify energy-intensive operations for better targeted optimization. These optimizations will ultimately lead to longer battery life for the device and a better user experience.

3) Call-graph (CG) generation: To detect the uses of the SQLite API in an iOS app, EC generates its CG using the Class-Hierarchy Analysis (CHA) [33] in SWAN [34]. We use CHA for a sound CG to provide developers with energy estimation for the largest number of potential execution paths that may execute at runtime.

4) API Call Methods Identification: EC traverses the CG to identify method calls in the code that perform SQLite operations. For each method that performs an SQLite operation, EC stores a method-to-API mapping for it, which consists of: the method name, the SQLite operation that it performs, source line at which the API is invoked, and the operation frequency. Figure 2 shows a sample iOS app that we will use to illustrate what method-to-API mappings are and how EC uses them to calculate E-factor. In this app, the primary methods that perform SQLite operations are `DB.update(..)`, `DB.insert(..)`, and `DB.read(..)`.

To create method-to-API mappings, EC first identifies the methods that call these primary methods. In class `A`, EC detects that `A.foo()` invokes `DB.insert()` (Line 32). Therefore, EC creates a method-to-API mapping of `A.foo() -> Insert @ Line 32`. EC also detects that `A.qux()` invokes `DB.insert()` twice (Line 36 and Line 37). Therefore, EC creates two method-to-API mappings of `A.qux() -> Insert @ Line 36` and `A.qux() -> Insert @ Line 37`. In class `B`, EC detects that `B.bar()` invokes `DB.update()` in a loop (Line 44). Therefore, EC creates a method-to-API mapping `B.bar() -> Update* @ Line 44` where `*` represents an operation in a loop. EC

```
30 class A {
31   func foo() {
32     DB.insert(...)
33   }
34
35   func qux() {
36     DB.insert(...)
37     DB.insert(...)
38   }
39   ...
40 }
```

(a) Class A

```
41 class B {
42   func bar() {
43     for k in 1 ... n:
44       DB.update(...)
45   }
46   ...
47 }
```

(b) Class B

```
48 class C {
49   func baz() {
50     B.bar()
51     A.foo()
52   }
53   override func tap(){
54     DB.read()
55   }
56   ...
57 }
```

(c) Class C

Fig. 2: Three Swift classes: `A`, `B`, and `C` in a sample iOS app that use `DB` to perform SQLite API operations.

identifies such operations by checking if an operation is enclosed in a `for` loop, `while` loop, or enclosed in a UI event, such as tapping/swiping a screen or touching a UI button. While a UI event is not considered a loop in the traditional sense, it may be triggered multiple times by the user. Therefore, for energy estimation, we consider a UI event as code that may execute in a loop. This assumption is essential for measuring the energy consumption of an app because the more frequently an event is triggered the more energy it consumes. In class `C`, EC detects that `C.baz()` invokes `B.bar()` (Line 50), which then invokes `DB.update()` in a loop. Because of this call chain, EC creates a method-to-API mapping `C.baz() -> B.bar() -> Update* @ Line 50`. EC also detects that `C.baz()` invokes `A.foo()` (Line 51), which then invokes `DB.insert()`. Given this call chain, EC creates a method-to-API mapping `C.baz() -> A.foo() -> Insert @ Line 51`. Class `C` has an additional function `tap()` that is a user input event and it invokes `DB.read()`. Therefore, EC creates a method-to-API mapping `C.tap() -> Read* @ Line 54`.

5) E-factor Calculation: Using the SQLite energy profile from Table I, EC identifies the energy consumption of each method-to-API mapping. The energy consumption of mappings that contain a `*` is represented by the range of energy consumption values in the energy profile for several number of

TABLE II: Energy consumption (in joules) of the method-to-API mappings for the sample app in Figure 2. The value "–" means "not applicable", and we use it for operations that are not in a loop.

		# of Executions							
Method-to-api mappings	Operation	$\text{Base}_{\text{exec}}$	10^1	10^2	10^3	10^4	10^5	10^6	10^7
`A.foo()-> Insert@Line 32`	Insert	0.0024	–	–	–	–	–	–	–
`A.qux()-> Insert@Line 36`	Insert	0.0024	–	–	–	–	–	–	–
`A.qux()-> Insert@Line 37`	Insert	0.0024	–	–	–	–	–	–	–
`B.bar()-> Update*@Line 44`	Update*	0.0025	1.20	1.23	2.66	25.14	249.33	2,494.38	23,677.88
`C.baz()->B.bar()-> Update*@Line 50`	Update*	0.0025	1.20	1.23	2.66	25.13	249.33	2,494.38	23,677.88
`C.baz()->A.foo()-> Insert@Line 51`	Insert	0.0024	–	–	–	–	–	–	–
`C.tap()-> Read*@ Line 54`	Read*	0.0002	1.61	1.73	1.78	2.28	15.53	184.99	1,877.99

TABLE III: E-factor values (in joules) of the aggregated methods from Table II. The value "–" means "not applicable", and we use it for operations that are not in a loop.

	# of Executions							
Method	$\text{Base}_{\text{exec}}$	10^1	10^2	10^3	10^4	10^5	10^6	10^7
`A.foo()`	0.0024	–	–	–	–	–	–	–
`A.qux()`	0.0048	–	–	–	–	–	–	–
`B.bar()`	0.0025	1.20	1.23	2.66	25.14	249.33	2,494.38	23,677.88
`C.baz()`	0.0049	1.20	1.23	2.66	25.14	249.33	2,494.38	23,677.88
`C.tap()`	0.0002	1.61	1.73	1.78	2.28	15.53	184.99	1,877.99
Total for App	0.0148	4.01	4.19	7.10	52.55	513.99	5,173.75	49,233.75

iterations. For example, if a mapping has a `Read*` operation, its energy consumption is 1.61 for 10^1 iterations, 1.73 for 10^2 iterations, etc. The energy consumption of a mapping that is not in a loop is the $\text{Base}_{\text{exec}}$ cost in the energy profile. Table II shows the energy consumption of each method-to-API mapping in the sample app from Figure 2. The table shows the method that performs an SQLite operation, its $\text{Base}_{\text{exec}}$ cost, and the energy consumption of executing that operation in a loop 10^1–10^7 times.

To calculate the E-factor value for each method, we aggregate and add up the energy consumption values from Table II by method name. Table III shows the final aggregated values. To calculate the E-factor of the complete API usage, we take the sum of the E-factor values of all methods. Table III shows that, our sample app may consume 0.0148 (10^0 iterations) to $49,233.75$ (10^7 iterations) joules depending on its SQLite usage.

Using EC, developers can calculate E-factor after each instance of code modification, which may be presented to them in the form of a table similar to Table III. This table displays the energy cost of a specific API usage for a method and for the entire app. EC is open-source and is freely available online [20].

IV. Evaluation

With E-factor, we evaluate these research questions:

RQ1: Can E-factor be used to compare the energy consumption difference between the versions of an app in the context of an API's usage?

RQ2: Can E-factor's fine-grained method-level energy estimates be used to compare the energy consumption of the methods of an app?

RQ3: Does our automated program (EC) accurately estimates the E-factor of real-world apps?

A. Dataset Preparation

To answer our research questions, we first select real-world apps and calculate their E-factor.

1) App Selection: To select apps for evaluation, we search GitHub [35] applying the following inclusion and exclusion criteria on all available iOS apps:

- Include all iOS projects written in Swift, because it is the currently supported language for CG generation in our underlying tool SWAN.
- Include projects that implement the SQLite API by searching for "`import sqlite3`" within the codebase.
- Exclude projects that are libraries, APIs, or incomplete. To ensure this criterion, we selected projects that have a user-interface code structure. We recognize that structure by searching for a UI storyboard or an `AppDelegate`.
- Exclude projects that have less than two commits in their repository, because the main use case for E-factor is comparing the energy consumption between app versions.
- Exclude projects that do not contain `.xcodeproj` or `.xcworkspace` file extension. This exclusion makes sure that the projects are executable.
- Exclude projects built for iOS platform < 9 because our underlying energy measurement infrastructure (iGreenMiner) supports apps built for iOS $>= 9$.

After applying our selection criteria, we ended up with 59 real-world apps. We group these apps into three categories (entertainment, healthcare, and utility), while some apps remain uncategorized because either their documentation is missing or their commit messages are not clear enough to

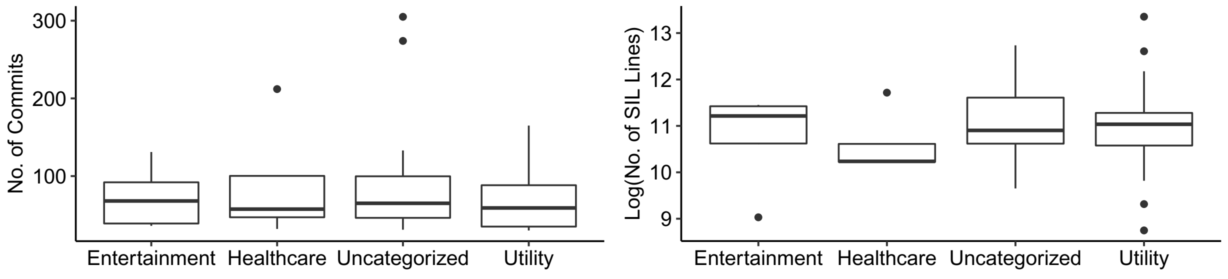

Fig. 3: Information about the main app categories for the real-world apps in our dataset.

infer their category. Figure 3 shows the number of commits and the number of SIL lines in each app in our dataset. The number of commits show how active the project development is, while SIL lines represent the project size.

2) E-factor Calculation: To detect methods that contain a use of SQLite, EC searches our dataset using case-insensitive regular expressions: `.*SELECT.*` for read operations, `.*UPDATE.*` for update operations, and `.*INSERT.*` for insert operations. EC then traverses the CG that SWAN generates for an app to identify those methods that invoke these SQLite methods. Using the identified method-to-API mappings, EC calculates the E-factor values of apps and their methods as explained in Section III-C5. Figure 4 shows the E-factor value for each app category for 10^0–10^7 executions. The figure also shows that the rank of app categories by E-factor changes across different number of iterations. Out of the 59 apps, there were 8 apps with zero E-factor value, because the SQLite operations in these apps are never invoked by a method. We could not estimate the E-factor of 3 additional apps, because they either contained incomplete code or had a third-party API dependency that we could not resolve to build the app. It took an average time of 12.44 milliseconds to calculate the E-factor for each of the 56 real-world apps, where the majority of this time was spent in call-graph traversal.

B. E-factor for Comparing Versions (RQ1)

1) Setup: In RQ1 we investigate if developers can use E-factor to compare the energy consumption of an API's usage between the various versions of an app.

a) Commit Selection: To consider an app from our dataset for RQ1, it has to be supported on our energy measurement setup using iGreenMiner by following this criteria: (1) should have a test-suite with UI tests, (2) should not depend on a third party web API to execute (e.g., Firestore), (3) should not require login credentials from a web service that we do not have access to, (4) should not run into runtime errors, (5) and should have an API use inside a loop or a UI event. Applying this criteria, we have 3 apps to investigate:

- CarTrack [36]: This app is used to register people with their profiles for record keeping. We identified 10 `insert` and 7 `select` operations in this app.
- Inventario [37]: This app is used for keeping an inventory list. We identified 8 `insert` and 5 `select` operations in this app.
- Planner [38]: This app is a doodle note taking app used to save a note for each calendar day. We identified 2 `insert`, 3 `select`, and 1 `update` operations in this app.

We then extracted 34 commits from CarTrack, 83 commits from Inventario, and 35 commits from Planner to calculate their E-factor. We could not build 16/152 commits because their code was insufficient to build the app. Eventually, we calculated E-factor for 136 commits in total. Figure 5 shows the app-level E-factor of each commit when an SQLite use would execute 10^4 times. The figure presents how E-factor evolves over time based on the SQLite usage in an app. Calculating E-factor for all commits was relatively straightforward because it does not require any test case execution. However, gathering hardware-based energy measurements (i.e., ground-truth) for all commits was unrealistic. This is because, for each commit, we would need to instrument it, then generate and execute its test cases. Following the methodology of Romansky *et al.* [39] for effectively mining commits to measure energy consumption, we selected commits whose SQLite use change over time. Therefore, we chose 16 commits for hardware-based energy measurement (highlighted in red boxes along the x-axis in Figure 5). The SHAs for these commits are:

- CarTrack: `2222255` and `b77d9b2`.
- Inventario: `e60848f`, `56af7e2`, `5b36854`, `5a6af9e`, `3479dee`, `6a48c89`, `6d6c717`, and `273890a`.
- Planner: `bc1a801`, `6319599`, `5e97690`, `ed78820`, `97d0057`, and `39714d2`.

b) Commit Instrumentation: To execute each commit on iGreenMiner, we had to manually configure the dependencies of each commit, especially older commits that rely on obsolete dependencies. This instrumentation required instrumenting the UI navigations and authentications in the apps to override business rules and methods such that all SQLite operation methods would execute within a single test-case execution. For instrumentation, we chose 10^4 iterations for the loops and UI events having SQLite operations because this is where E-factor starts to noticeably increase (Figure 4). It was also unrealistic to measure for more than 10^4 iterations, because at

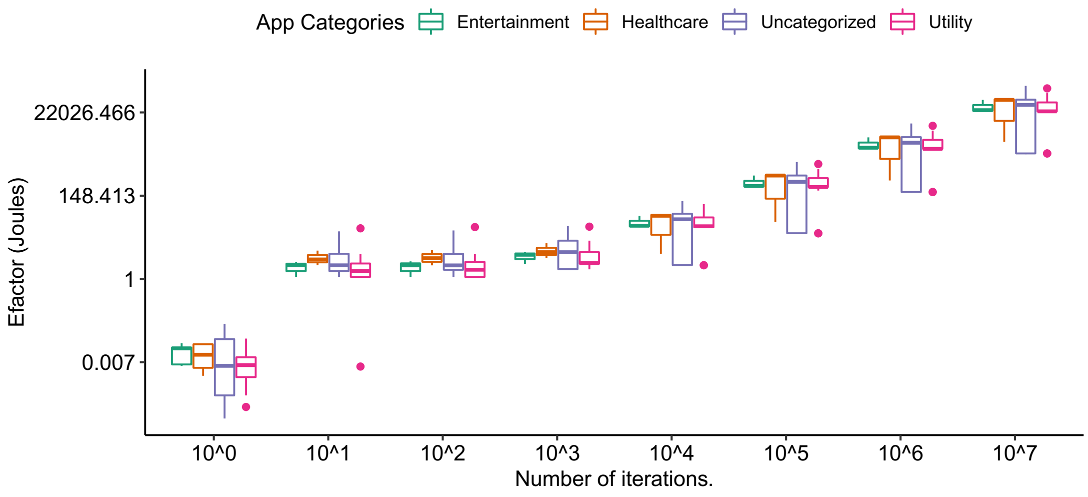

Fig. 4: The E-factor values of each app category. The range 10^0–10^7 is the number of times that each app may execute its SQLite operations during runtime.

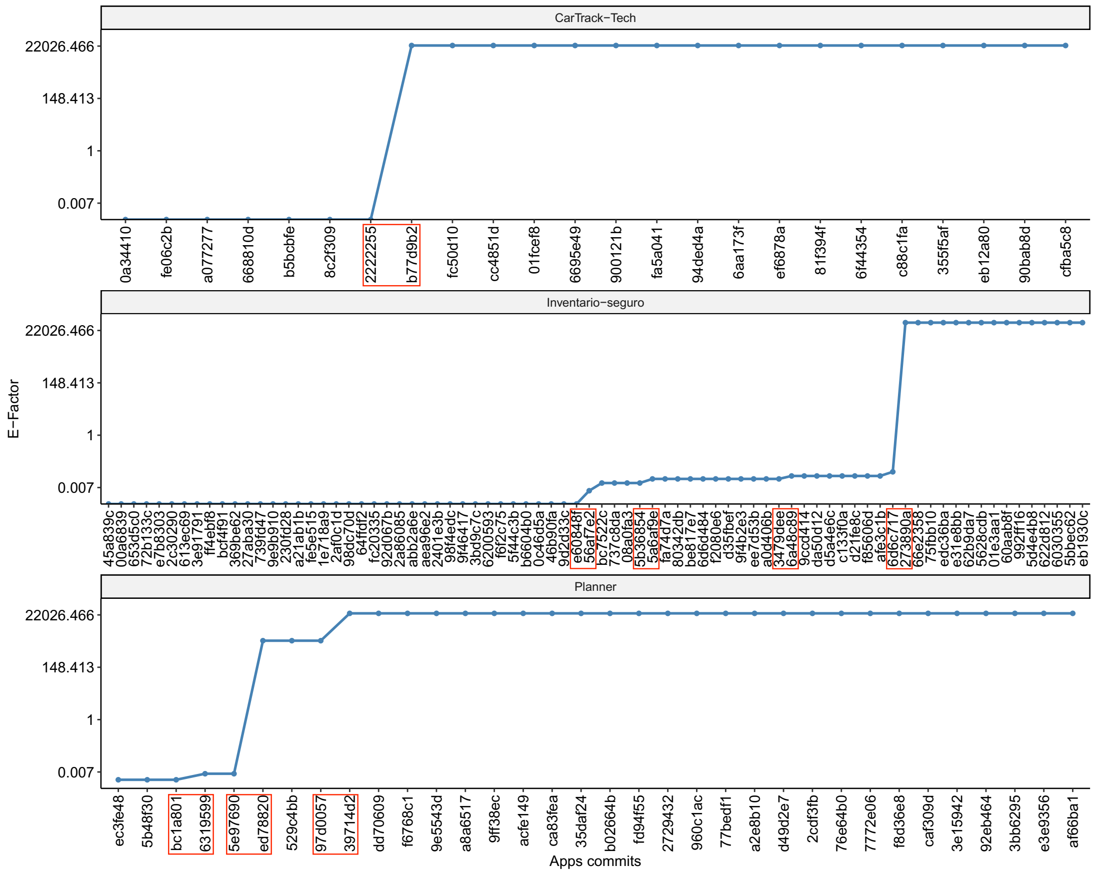

Fig. 5: E-factor calculated from the commits of CarTrack, Inventario, and Planner. The commits are ordered left-to-right in sequence over time (i.e., oldest to latest commit). Commits whose E-factor change over time are highlighted in a red box. The number of iterations assumed for the SQLite usages that are in a loop is 10^4.

that point the highest energy consuming app, Inventario, takes ≈ 57 hours to execute.

c) Hardware Energy Measurement: To accurately measure the energy consumption of each commit, we ran the 16 instrumented versions on iGreenMiner multiple times. We execute the versions that took less than an hour 10 times, and the ones that took an hour or more 5 times. We use these versions' energy measurements and compare them with their E-factor.

2) Results: To compare E-factor to hardware-based energy measurements of the commits, we use the Spearman's Rank Correlation with the Bonferroni correction method [40]. The Spearman's Rank Correlation reports $\rho = 0.60$ with a p-value < 0.05, indicating a significant correlation between E-factor and hardware-based energy measurements of the commits. Using Hopkin's scale [41], $\rho = 0.60$ indicates that if a commit has a lower E-factor than another, then it is more energy-efficient due to the large positive correlation between E-factor and energy measurements. Given this significant positive correlation, E-factor is a reliable indicator for developers to determine if their latest code change related to the SQLite API usage would result in an increase or decrease in energy consumption compared to a previous version of their app.

C. E-factor for Fine-grained Energy Review (RQ2)

For a given API usage in an app, if a developer discovers that their app consumes too much energy, they may use E-factor to compare the energy consumption of different methods within the app that use that API. In this research question, we investigate whether E-factor provides this information to developers to help them focus their optimization efforts on specific methods.

1) Setup: For this investigation, we consider methods of the same set of apps that were used in RQ1: CarTrack, Inventario, and Planner. We have already calculated the E-factor of these apps and their methods. To collect the hardware-based energy measurements (i.e., ground truth) of these methods though, we need to first instrument the apps such that each method executes in isolation.

a) Method Selection: CarTrack has 8 methods that perform SQLite operations. For evaluation, we consider `CountryRepo.createCountryData()` that performs an `insert` in a loop. From the remaining methods that perform a single `select`, we could only instrument `Country-Repo.init(...)` to execute in isolation. Inventario has 6 methods that perform SQLite operations. `LoginVC-.createDummyUsers()` performs 5 `insert` operations and 1 `select`. `mostrarVC.InsertRollos()` and `mostrarVC-.InsertRegistros(..)` perform `insert` in a loop and 1 `select`. `VC.dummys()` performs 1 `insert` and 1 `select`, while `DBUsuarioHelper.insert(..)` and `LoginVC.do-Login(..)` perform 1 `select`. We consider all these methods in our evaluation. Planner has 5 methods that perform SQLite operations. Out of the four methods that perform `insert` and `select` once; we consider `CanvasVC.createNew-Canvas()` and `CanvasVC.loadCanvas()`. Finally, we consider `CanvasVC.drawingDidChange(...)` that performs a `select` and an `update` in a loop.

b) App Instrumentation: We created 2 instrumented versions for CarTrack, 6 instrumented versions for Inventario, and 3 instrumented versions for Planner. Similar to RQ1, each instrumented version executes a method 10^4 times if it would occur in a loop or a UI event, and once otherwise.

c) Hardware Energy Measurement: To collect the ground truth, we executed each instrumented version 10 times on iGreenMiner and collected their energy consumption measurements. Since each instrumented version represents a method, to compare a method's energy consumption with its E-factor, we compare the instrumented versions' energy measurements with the representative method's E-factor value.

2) Results: Table IV compares E-factor to the ground truth of each method. The table shows that E-factor reports the highest values for the most energy consuming methods in an app: `CountryRepo.createCountry-Data()` in CarTrack, `mostrarVC.InsertRegistros(..)` and `DBUsuarioHelper.insert(..)` in Inventario, and `CanvasVC.drawingDidChange(...)` in Planner.

To determine the relationship between E-factor and hardware-based energy measurements, we used the Spearman's Rank Correlation with the Bonferroni correction method. The Spearman's Rank Correlation reports $\rho = 0.59$ with a p-value < 0.05, indicating a significant correlation between E-factor and hardware-based energy measurements of the evaluated methods. Using Hopkin's scale [41], $\rho = 0.59$ indicates that if a method has a lower E-factor than another, then it is more energy-efficient due to the large positive correlation between E-factor and energy measurements. Given this significant positive correlation, developers may reliably use E-factor to compare the energy consumption of API usage within methods in their code without executing them.

D. E-factor's Automation Capability (RQ3)

In RQ3, we want to verify if EC calculates E-factor accurately. To verify the accuracy, we compare EC with the manual computation of E-factor.

1) Setup: For this evaluation, we used 12 apps (3 from each category) with the minimum, average, and maximum automatically calculated E-factor values:

- Entertainment: InstagramProjectIos, Instant, and Priyo Movies
- Healthcare: HealthKitTest, iOS-App-Project, and careapp
- Utility: Dulce, Storage, and Planner
- Uncategorized: Tourus, ThisAppTeam, and HybridApp

To manually compute E-factor (i.e., ground truth), we first traversed through the code files of each app. We then identified method-to-API mappings that are reachable to execute during runtime.

2) Results: To compare the automatically calculated EC E-factor values with the manually calculated E-factor, we compute the mean absolute percentage error (MAPE) [42]

TABLE IV: Comparing E-factor with hardware-based energy measurements for methods from real-world apps. The number of iterations assumed for the SQLite usages that are in a loop is 10^4.

App	Method	Operation	Ground Truth (joules)	E-factor (joules)
CarTrack	`CountryRepo.createCountryData()`	`insert*`	62.27	22.2504
CarTrack	`CountryRepo.init(...)`	1 `select`	36.93	0.0002
Inventario	`VC.dummys()`	1 `insert` + 1 `select`	4.03	0.0027
Inventario	`DBUsuarioHelper.insert(..)`	1 `select`	2.85	0.0002
Inventario	`LoginVC.createDummyUsers()`	5 `insert` + 1 `select`	3.25	0.0124
Inventario	`LoginVC.doLogin(..)`	1 `select`	3.37	0.0002
Inventario	`mostrarVC.InsertRegistros(..)`	`insert*` + 1 `select`	8446.95	22.2504
Inventario	`mostrarVC.InsertRollos()`	`insert*` + 1 `select`	3899.09	22.2504
Planner	`CanvasVC.createNewCanvas()`	1 `insert`	1.63	0.0025
Planner	`CanvasVC.loadCanvas()`	1 `select`	1.74	0.0002
Planner	`CanvasVC.drawingDidChange(...)`	`update*` + `select*`	11.30	27.4114

between both. MAPE shows that the percentage error in E-factor values is 16.84%, indicating that EC calculates E-factor of real-world apps within 20% of the mean absolute error margin. This margin is acceptable because, in energy estimation models, MAPE may vary from 5% to 1,000% depending on the number of apps used to train the model [9].

To investigate the reason behind the error, we compare the manually detected method-to-API mappings with EC detected mappings. Upon this investigation, we found that EC identified the mappings with a precision of 0.91 and a recall of 0.84 with an F1 score of 0.87. These measurements show that EC is less likely to provide false positives but may miss some true positives.

Upon further investigation, we have also identified the following reasons for the occurrence of false positives (i.e., reporting a mapping that does not exist) and false negatives (i.e., not reporting a mapping that exists):

- EC could not identify UI actions in input apps if the responsible action implements a method other than `tapGesture` or `onClick` from the Swift UI framework.
- Some identified execution paths from the CG are infeasible because they do not execute at runtime.
- The underlying framework that generates the CG (SWAN) does not support recursion or Swift structs.

V. Related Work

In this section, we briefly discuss prior work that is closely related to our approach in terms of methodology and results.

Lyu *et al.* [43] empirically investigated the affect of SQL anti-patterns on the energy consumption of smartphone apps and discovered that SQL operations performed in a loop may increase an app's energy consumption considerably, and suggest that developers should use join SQL queries instead of loops when possible. Their work, however, is a recommendation model for SQL usage, unlike our work that estimates the energy consumption of any API's usage. Hao *et al.* [8] used static analysis to provide accurate energy measurements at line-level granularity. However, their presented approach requires an app's runtime information and the target smartphone components' energy profile from its manufacturers. It is impractical to acquire a manufacturer's built energy profile as it could not be possible when the owner holds proprietary rights over their components, such as Apple Inc. On the other hand, our approach does not require any runtime information or test case execution, and acquiring the energy profile in our case is straightforward because anyone owning a smartphone and an energy measurement instrument can obtain their selected API's energy profile.

Li *et al.* [18] empirically investigated 405 real-world Android apps and found some interesting facts about API usage's energy consumption in smartphone apps. They discovered that API events consume 85% of the energy in an app compared to developer-written instructions (e.g., branch or arithmetic instructions). They also discovered that SQLite queries are the third most energy-consuming factor in apps following Network and Camera related operations. This finding signifies the importance of API usage's energy estimation in general and SQLite usage in specific. Finally, they discovered that operations in a loop represent 41.1% (average) of an app's energy consumption when in a non-idle state. Moreover, loops that use an API may consume 6–41% more energy than loops that do not use an API. This finding further signifies the importance of estimating the energy consumption of API usages in loops and UI events.

Similar to our work, Jabbarvand *et al.* [44] focus on measuring the relative energy consumption of the apps to rank different apps by energy cost. However, their approach involves dynamic analysis and it executes test cases, whereas ours does not require any runtime information, thus it relieves app-developers from the burden of test case execution.

Hasan *et al.* [45], Oliveira *et al.* [46], and Pereira *et al.* [47], [48] use static analysis and profile the energy consumption of the Java Collections API to help developers implement the most energy-efficient collection in their code. However, we use static analysis and energy profiles differently. Instead of optimizing a developer's code, we aim to provide developers with a perspective on how expensive, in terms of energy consumption, the methods or a version of their app could become during runtime.

VI. Threats to Validity

In this section, we discuss the threats to the validity of our results and how we addressed them.

A. Construct Validity

To find call sites that use SQLite queries, we construct a CG using CHA [33]. There are many other CG construction algorithms such as RTA [49] and XTA [50] that are faster than CHA, and each of these algorithms may yield different results for E-factor. However, we have chosen CHA to over-approximate the calculated energy estimates i.e., provide developers with maximum possible energy consumption.

For hardware-based energy measurements, we used iGreen-Miner that supports iPhone 11 running iOS 13.4.1. However, a different smartphone device and operating system may yield a different energy profile for the SQLite API. Therefore, our energy profile and results may not generalize over all smartphone platforms and devices that run them.

B. Internal Validity

To measure the energy profile of the SQLite API, we pre-populated the DB records. However, our DB records may not represent real-world scenario because we keep the initial database empty each time before executing the `insert` benchmarks and pre-populate 10^7 records before executing the `select` and `update` benchmarks. Having a different number of records affects the runtime performance of SQLite differently [51] and realistically, users perform SQLite operations on a variable number of records. For better SQLite energy profiling, future work may adopt parallelization to run multiple instances of a DB on various phones, where each DB will have a random number of records, columns, and indexes. Regardless, we believe that the relative difference between `insert`, `select`, and `update` may stay consistent when a DB has the same structure and number of records to start with. However, this claim requires an empirical evaluation that is out of the scope of this paper.

We have designed our approach to consider only two usage-contexts for SQLite operations: those that occur inside a loop and those that occur outside of it. However, SQLite operations may occur in many other contexts, such as inside a branch condition or an asynchronous thread. In this study, we assume that all branches execute and we do not consider all other possible contexts. Therefore, we are unable to provide a best-case or even an average-case energy estimate. Instead, we provide worst-case estimates only.

The SIL files that we analyze represent the developer's code only, and therefore our E-factor values are confined to the developer's code. We do not calculate the E-factor of third-party libraries on which an app may depend. However, this can be done by generating and analyzing the SIL files for the external libraries. Nonetheless, this is out of the scope of this paper, and we leave it for future work.

In Table IV, the E-factor value of `CanvasVC.drawingDidChange(...)`, in which two different SQLite operations in a loop run in combination, is larger than the ground-truth value. This inaccuracy might occur because we had separately profiled the energy consumption of `update` and `select`, and we never profiled their combinatorial effect. Operations, when executed in combination, may follow several architecture-level optimizations. Therefore, for more accurate E-factor values, different operations combinations should also be profiled.

C. External Validity

Our SQLite API energy profile and results are relevant to apps built for iOS 13.4.1. To claim the generalization of our results over all iOS apps, the energy profile has to be obtained using other iOS versions.

Our results are limited to the Swift SQLite API, which exploits the processing subsystem of a smartphone. Exploring additional APIs that use other energy-extensive subsystems (i.e., networking, location, and graphics) is out of the scope of this paper. Nevertheless, our work is a foundation for API-usage energy estimation and facilitates the evaluation of other APIs by providing a generic step-wise methodology.

E-factor provides the API usage energy estimates while assuming that the API use will execute for a range of iterations. Future work may better estimate the runtime iterations of loops through static analysis techniques that use abstract interpretation [52] or path dependency automation [53].

VII. Conclusion

App developers use hardware-based measurement tools or software-based estimation models to measure their apps' energy consumption. However, the current hardware and software-based measurement and estimation techniques are cumbersome because they require developers to own a smartphone and generate and execute test cases. As a solution, in this paper, we have proposed a static-analysis-based approach that can estimate the energy consumption (E-factor) of API usage in an app without the need to generate and execute test cases. We evaluate our technique on real-world apps that use the SQLite API and find out that E-factor is a good estimator to compare the relative energy consumption of API use between the versions of an app and within the methods of an app. Our proposed approach enables energy estimates to be made at compile time. Additionally, our approach provides an opportunity for integration with existing Integrated Development Environments (IDE)s or Continuous Integration/Continuous Deployment (CI/CD) pipelines, providing automated energy testing as a part of the development process. Such integration would reduce the costs associated with energy testing and improve the overall development process by identifying and fixing energy-inefficient code early on in the development cycle. To improve the estimation of E-factor, it is recommended to include energy profiles of additional APIs in the computation. Therefore, we encourage the research community to contribute to our public GitHub repository [20] by adding more energy profiles.

Acknowledgement

This research was supported by the Discovery Grant program of Natural Sciences and Engineering Research Council of Canada (NSERC) .

References

[1] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, “What do mobile app users complain about?” *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[2] G. Pinto, F. Castor, and Y. D. Liu, “Mining questions about software energy consumption,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 22–31.

[3] J. M. Hirst, J. R. Miller, B. A. Kaplan, and D. D. Reed, “Watts up? pro ac power meter for automated energy recording,” 2013.

[4] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, “Greenminer: A hardware based mining software repositories software energy consumption framework,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 12–21.

[5] X. Li, X. Zhang, K. Chen, and S. Feng, “Measurement and analysis of energy consumption on android smartphones,” in *2014 4th IEEE International conference on information science and technology*. IEEE, 2014, pp. 242–245.

[6] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, “Software-based energy profiling of android apps: Simple, efficient and reliable?” in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 103–114.

[7] G. Pinto and F. Castor, “Energy efficiency: a new concern for application software developers,” *Communications of the ACM*, vol. 60, no. 12, pp. 68–75, 2017.

[8] S. Hao, D. Li, W. G. Halfond, and R. Govindan, “Estimating mobile application energy consumption using program analysis,” in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 92–101.

[9] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, “Greenscaler: training software energy models with automatic test generation,” *Empirical Software Engineering*, vol. 24, no. 4, pp. 1649–1692, 2019.

[10] M. B. Kjærgaard and H. Blunck, “Unsupervised power profiling for mobile devices,” in *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2011, pp. 138–149.

[11] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, “The mälardalen wcet benchmarks: Past, present and future,” in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[12] T. Nolte, H. Hansson, and C. Norstrom, “Probabilistic worst-case response-time analysis for the controller area network,” in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.* IEEE, 2003, pp. 200–207.

[13] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, “Smartphone energy drain in the wild: Analysis and implications,” in *Proceedings of the 2015 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems*, ser. Sigmetrics ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 151–164. [Online]. Available: https://doi.org/10.1145/2745844.2745875

[14] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[15] L. C. Briand, “A critical analysis of empirical research in software testing,” in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 2007, pp. 1–8.

[16] S. McMaster and A. M. Memon, “An extensible heuristic-based framework for gui test case maintenance,” in *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 251–254.

[17] M. Mirzaaghaei, F. Pastore, and M. Pezze, “Automatically repairing test cases for evolving method declarations,” in *2010 IEEE International Conference on Software Maintenance*. IEEE, 2010, pp. 1–5.

[18] D. Li, S. Hao, J. Gui, and W. G. Halfond, “An empirical study of the energy consumption of android applications,” in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 121–130.

[19] AppleInc., “What’s new in energy debugging,” 2018. [Online]. Available: https://developer.apple.com/videos/play/wwdc2018/228/?time=265

[20] A. A. Bangash, “Sqlite energy profiles with benchmarks, and e-factor calculator,” Mar 2023. [Online]. Available: https://zenodo.org/record/7262615

[21] A. A. Bangash, D. Tiganov, K. Ali, and A. Hindle, “Energy efficient guidelines for ios core location framework,” in *Proceedings of the 2021 International Conference on Software Maintenance and Evolution (ICSME)*, 2021, inproceedings, pp. 1–12. [Online]. Available: http://softwareprocess.ca/pubs/bangash2021ICSME-igreenminer.pdf

[22] M. Solutions, “High voltage power monitor,” 2021. [Online]. Available: https://www.msoon.com/

[23] AppleInc., “Apple’s integrated development environment (ide),” 2022. [Online]. Available: https://developer.apple.com/xcode/

[24] A. Puvvala, A. Dutta, R. Roy, and P. Seetharaman, “Mobile app developers’ platform choice model,” in *2016 49th Hawaii International Conference on System Sciences (HICSS)*. IEEE, 2016, pp. 5721–5730.

[25] M. Fitzgerald, “Android vs. ios app development: Which is the better choice for your business in 2019?” Oct 2019. [Online]. Available: https://bit.ly/3nbOCkS

[26] M. H. Goadrich and M. P. Rogers, “Smart smartphone development: ios versus android,” in *Proceedings of the 42nd ACM technical symposium on Computer science education*, 2011, pp. 607–612.

[27] D. R. Hipp, “Sqlite docs,” 2000. [Online]. Available: https://www.sqlite.org/about.html

[28] E. Noei, F. Zhang, and Y. Zou, “Too many user-reviews! what should app developers look at first?” *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 367–378, 2019.

[29] SQLite, “Features of sqlite,” 2022. [Online]. Available: https://www.sqlite.org/features.html

[30] S. Celis, “Sqlite swift wrapper,” 2022. [Online]. Available: https://github.com/stephencelis/SQLite.swift

[31] SQLite, “Well-known users of sqlite,” 2022. [Online]. Available: https://www.sqlite.org/famous.html

[32] AndroidDoc, “Save data in a local database using room,” 2022. [Online]. Available: https://developer.android.com/training/data-storage/room

[33] J. Dean, D. Grove, and C. Chambers, “Optimization of object-oriented programs using static class hierarchy analysis,” in *European Conference on Object-Oriented Programming*. Springer, 1995, pp. 77–101.

[34] D. Tiganov, J. Cho, K. Ali, and J. Dolby, *SWAN: A Static Analysis Framework for Swift*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1640–1644. [Online]. Available: https://doi.org/10.1145/3368089.3417924

[35] MicrosoftCorp., “A code hosting platform for version control and collaboration.” 2022. [Online]. Available: https://github.com/

[36] R. Correia, “Car track tech challenge app,” 2021. [Online]. Available: https://github.com/ricardo-correia/CartrackTechChallenge

[37] J. Margarita, “Inventario seguro app,” 2021. [Online]. Available: https://github.com/ProyectoIntegrador2018/inventario_seguro

[38] A. Clare, “Planner app,” 2021. [Online]. Available: https://github.com/clare228/Planner

[39] S. Romansky and A. Hindle, “On improving green mining for energy-aware software analysis,” in *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, 2014, pp. 234–245.

[40] J. H. Zar, “Spearman rank correlation,” *Encyclopedia of biostatistics*, vol. 7, 2005.

[41] W. G. Hopkins, “A new view of statistics,” 2002.

[42] A. De Myttenaere, B. Golden, B. Le Grand, and F. Rossi, “Mean absolute percentage error for regression models,” *Neurocomputing*, vol. 192, pp. 38–48, 2016.

[43] Y. Lyu, A. Alotaibi, and W. G. Halfond, “Quantifying the performance impact of sql antipatterns on mobile applications,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 53–64.

[44] R. J. Behrouz, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, “Ecodroid: An approach for energy-based ranking of android apps,” in *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*. IEEE, 2015, pp. 8–14.

[45] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, “Energy profiles of java collections classes,” in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 225–236.

[46] W. Oliveira, R. Oliveira, F. Castor, G. Pinto, and J. P. Fernandes, “Improving energy-efficiency by recommending java collections,” *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–45, 2021.

[47] R. Pereira, P. Simão, J. Cunha, and J. a. Saraiva, “Jstanley: Placing a green thumb on java collections,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software*

Engineering, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 856–859. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3238147.3240473

[48] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 46–57.

[49] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996, pp. 324–341.

[50] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2000, pp. 281–293.

[51] M. J. Carey and D. Kossmann, "On saying "enough already!" in sql," *SIGMOD Rec.*, vol. 26, no. 2, p. 219–230, jun 1997. [Online]. Available: https://doi.org/10.1145/253262.253302

[52] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009, pp. 136–146.

[53] X. Xiaofei, "Static loop analysis and its applications," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1130–1132.