Exploring Potential and Feasibility of Binary Code Sharing in Mobile Computing

Chao Wu, *Member, IEEE*, Lan Zhang, *Member, IEEE*, Zhenhua Li, *Member, IEEE*, Qiushi Li, and Yaoxue Zhang, *Senior Member, IEEE*

Abstract—While tremendous growing mobile apps offer users rich services and functionalities, they also bring significant performance and energy issues. Code sharing is promising to address these issues, but existing application-level code sharing is rather restrictive. This paper develops the a transparent machine code sharing for mobile devices, and presents its design, implementation, and deployment. SnapCode enables machine code sharing across a wide variety of commercial off-the-shelf Android devices. By sharing and running machine code, SnapCode can offer significant speed-ups: an average speed-up of 9.9X for one-time trial apps, and up to 120X in apps' regular uses. In addition, it can save more than 80 percent energy consumption.

Index Terms—Mobile computing, binary code sharing, system performance

1 Introduction

Motivation. The number of mobile apps has grown tremendously since the launch of app markets. Till June 2016, more than 2 million apps are available on both Google Play and Apple App Store. While the apps provide with rich services and attractive functions, users suffer from onerous system-on-chip (SoC) workloads and frequent data exchanges, which may lead to unexpected battery drain, undesirable data redundancy, and intolerable long service latency [1]. Although the performance issues are well known, there are only a few efforts to address it recently, among which the most promising philosophy is called code offloading. For example, Google Now and Nextbit Robin [2] have been offloading the workloads from end-user devices to their infrastructure servers. CloneCloud [3], COMET [4] and OGSI.NET [5] report the effect of code offloading on representative apps. Nevertheless, competition in today's mobile markets has led to numerous "walled-gardens", where developers build their own suites of applications that keep users within their ecosystems. Specifically, let's consider the YouTube app on Android platform, it can on average cost as many as 38 seconds to finish an installation or upgrade process on a Samsung S5 LTE-A smartphone (see Fig. 4). This essentially hinders the wide adoption of the state-of-the-art application-specific code offloading.

- *C. Wu, Q. Li, and Y. Zhang are with the Key Laboratory of Pervasive Computing, Ministry of Education Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: {chaowu, zyx}@tsinghua.edu.cn, lqs17@mails.tsinghua.edu.cn.*
- *L. Zhang is with the University of Science and Technology China, Hefei, Anhui 230009, China. E-mail: zhanglan@ustc.edu.cn.*
- *Z. Li is with the School of Software, Tsinghua University, Beijing 100084, China. E-mail: lizhenhua1983@tsinghua.edu.cn.*

Manuscript received 10 Oct. 2018; revised 10 June 2019; accepted 28 July 2019. Date of publication 1 Aug. 2019; date of current version 8 Mar. 2022. (Corresponding author: Chao Wu.)
Recommended for acceptance by B. Benatallah.
Digital Object Identifier no. 10.1109/TCC.2019.2932386

An alternative approach is to transparently intercept and optimize mobile computing across heterogeneous apps at the underlying OS layer, i.e., *sharing machine codes*. Although some examples using machine codes for cross-application optimization already exist over desktops [6], [7], [8], little is known about their feasibility and performance on mobile platforms as it is extremely challenging to address issues in both Java and native code compilation, i.e., runtime parameters passing and SoC's instruction set compatibility [9].

Our Approach. In this paper, we explore the potential and possibility of mobile machine code sharing, and present the design, implementation, and deployment experiences with SnapCode, a lightweight and flexible framework similar to Contiki [10] for heterogeneous Android devices. Fig. 1 sketches the main idea of SnapCode, where the ecosystematic issue and inefficient steps at application layer are avoided or mitigated by shifting the code sharing to the OS layer. In this way, we bridge machine codes on the cloud with workload over SoC, and enable any peer to incrementally download and run them directly. To maximize the benefits of mobile code execution, we target at not only low latency and minimum energy cost, but also efficient management of local user data [11], [12], [13], [14], [15]. By transferring user data to private cloud, SnapCode can achieve a clean use, i.e., no residual user data. In practice, by using SnapCode, mobile users can obtain up to 9.9X speed-up for one-time trial apps and 120X in apps' regular uses while saving as much as 80 percent battery life. Thus, there brings a significant and promising optimization, as Section 6 elaborates.

Through both experimental benchmarks and trace-driven measurements (Table 2), we find that compiling application codes into machine codes incurs 83 percent of time and 69 percent of energy overheads, while 86.5 percent of compilation are just for one-time trial app uses. In

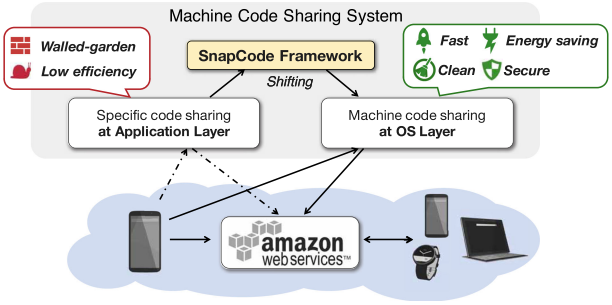


Fig. 1. Main idea of SnapCode: shifting the code sharing from the application layer to the OS layer to achieve a fast, clean and energy-saving experience, which brings transparent architecture innovation to users.

addition, more than 10 percent of mobile users are using the same most popular 20 apps (see Fig. 3). These findings demonstrate the great potential of sharing machine codes within the mobile device community, and inspire us to let a small number of Android devices generate input-to-execute bitcodes (machine codes), which are further shared by most peers and processed on their own SoC as native methods.

While SnapCode can bring dramatic experience improvements to Android community, its implementation may still face the instruction compatibility issue across diverse SoCs. Fortunately, ARM and Intel takes almost all market share in mobile processors [16], [17]. We deploy SnapCode atop a variety of mainstream Android devices including Samsung S3, S4, S5; LG Nexus 5; Moto Nexus 6 and 360 smartwatch; HUAWEI Nexus 6P and smartwatch to comprehensively evaluate the feasibility of our proposed design. As expected, via identifying the same instruction set with OS interface (Section 3), machine codes can be shared across heterogeneous Android platforms (Section 6.2), thus indicating the promising future of SnapCode. Moreover, we investigate the possibility of integrating the SnapCode framework into Android by partially re-compiling the AOSP project [18], which frees the users from the requirements of *super-user* (ROOT) privileges. In addition, a series of lightweight methods are leveraged to deal with other security vulnerabilities.

Specifically, in designing an efficient and compatible framework for Android machine code sharing, SnapCode targets four key goals:

1. We want to benefit users with significantly enhanced experiences, including lower latency, less energy overhead, and easy management of user data.
2. Machine code sharing could in principle address the compatibility issues caused by heterogeneous Android platforms, e.g., machine codes shared by a Sumsung Galaxy S5 can directly run on a Moto 360 watch.
3. We wish our design can be implemented within the scope of existing infrastructure and techniques, thus requiring minimum engineering efforts from developers.
4. Our design must provide users with comprehensive security protection by taking all potential vulnerabilities into consideration.

By sharing machine codes over existing Android ART mechanism, our evaluation shows that SnapCode can achieve a speed-ups of $9.9X$ for app install on new devices (i.e., do not have the app), $120X$ speed-ups comparing with default app launch (i.e., with reusable objects), and reduce over 80 percent battery life against with existing approaches.

TABLE 1
Data Collection from Baidu Inc.

Collection period	11/17/2014, 12/11-12/12 2014, 07/23-07/22/ 2015		
Unique users	90128	**Total requests**	3000M
Total off-the-shelf apps	343	**Common apps of the users**	37
HTTP traffic record size			330.2 GB

Contributions and Roadmap:

- Identifying key challenges in building a machine code sharing system for mobile community (Section 2).
- Design and implementation of SnapCode, built on key goals to address the challenges (Sections 3, 4, and 5).
- Real-world and trace-driven evaluation that demonstrates substantial improvement by SnapCode (Section 6).
- Using practical features to shape potential performance especially in vulnerable environments (Section 7).

2 Background and Challenges

This section analyzes common problems caused by today's mobile app uses (Section 2.1). We close this section by elaborating on the challenges of our work (Section 2.2).

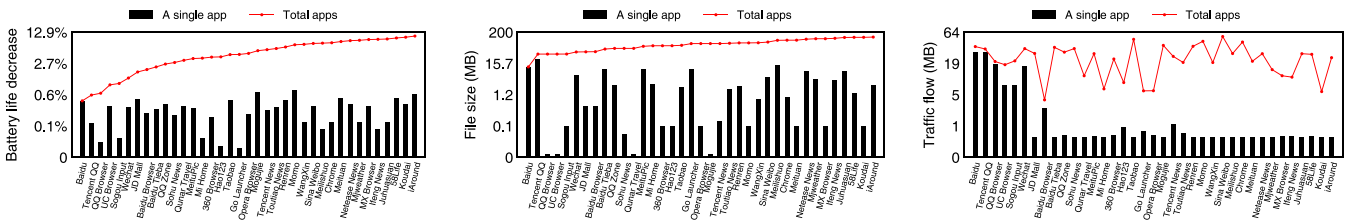
2.1 Background and Data Analysis

The past decade has witnessed a tremendous advance in mobile computing. A lot of efforts have been devoted to improve both mobile devices and the computing paradigm, but we still face many challenges to satisfy users' increasing demands for ideal app use experience. To explore the negative aspects, we analyze app use data of app requests over mobile HTTP traffic from 90,128 Baidu Security app [19] users,[1] which was recently collected from a major mobile Internet service provider of China, i.e., Baidu Inc. (Table 1). Indeed, Fig. 3 further demonstrates our statistics with the collected data, and find only 37 apps are considered as popular for more than 10 percent users installed those apps.

By replaying (i.e., re-request) the 37 app from Baidu Security server on a Samsung S5 LTE-A smartphone with all user records,[2] Fig. 2 summaries two key drawbacks requiring urgent optimization for user experience: *(1)* as shown in Fig. 2a, there is a prominently energy cost for experiencing apps (up to 13 percent battery life on average); *(2)* as shown in Fig. 2b, mobile OS, e.g., Android, can not remove all user-related files, often leaving device with increasing user data (up to 112 MB/day on average). In addition, Fig. 2c explores users' behavior characteristics as: on average, a user can daily consume 10 MB cellular data for installing or updating the 37 popular mobile apps. Such app uses account for more than 86.5 percent traffic excluding multimedia contents,

1. We randomly invited volunteers from $\sim 100M$ existing mobile users of the company and obtained informed consent from them by prominently informing them that full traces of their cellular traffic would be collected and analyzed.

2. In practice, mobile users often hold diverse devices, which can result in non-standard cellular data and energy measurements.



(a) Average energy cost for app trials (b) Average local data increment for app trials (c) Average traffic flow consumed to app trials

Fig. 2. Measurements of daily energy cost, local user data increment, and network traffic flow usage in experiencing the most popular 37 apps. While in most cases, users tend to try app likely.

which means in most cases users often make one-time trial decisions easily. These facts show that most energy, storage, traffic and computation are inefficient, and thus leave us a room to improve the mobile computing paradigm and optimize on-the-go user experience.

2.2 Challenges

The demands for better user experience call for both academic and industrial studies on improving computing paradigm, especially in mobile environment.

Root Cause. As analyzed, we reveal the root causes of nowaday's mobile apps' poor performance as: 1) high latency and huge energy overhead; 2) increasing user data and non-trivial cost for either maintenance, namely installation and version updating, or secure protection.

Less Pain, But "Walled-Garden" in Ecosystem. Code offloading has grown in popularity with the advent of cloud computing, e.g., Nextbit Robin [2], where many low-powered, well-connected computing elements could benefit from the computation of nearby server-class machines [3], [4], [20], [21]. We further explore the benefits of shifting from native execution to code offloading paradigm by measuring service latencies, including both loading and execution time, and energy cost for trying three popular apps more than 50 runs on three models of Samsung smartphones. Fig. 4 shows the relative speed-up factor (latency improvement) and energy cost of Cloud app and Off-loaded app thread codes with WiFi.[3] By offloading YouTube, the latency mitigates from 38s to 10s over a Google onHub with 802.11ac channel, which brings significant reductions to both latency and energy. Although promising to mitigate users' pain, the existing offloading methods are tailored for OS-specific application layer where the heterogeneity severely hinders the wide adoption of existing code offloading to commercial off-the-shelf (COTS) mobile devices. For example, although using the same OS kernel, the Samsung Pay app's code cannot be offloaded and processed to other Android platforms on the application layer.

Towards Better User Data Managing, But Can be Exhausted. Cloud app gives an insight to provide centralized maintenance where user-related files are cached on cloud, which can lead to no local user data. In practice, users subscribe to cloud services, download applications from cloud server, and then associated data with cloud storage, e.g., content servers [22]. Nevertheless, as code should be first compiled then can be run, obtaining apps from cloud often incurs higher latency and energy overheads than using code offloading. For example, we see that with better hardware support, e.g., the YouTube cases of S5 and S3 in Fig. 4, cloud app costs more latency (11 s) and energy (65 J). Thus, the overhead can be still exhausted for mobile users and urge for improvement.

Opportunity. Together, the above issues demonstrate the insufficiency of today's computing paradigm at the application layer, and motivate us to dig deeper in the OS layer for better solution, where the ecosystematic issues are avoided or mitigated. We observe that, most Android community can be powered by machine code programming as they use the same SoC instructions [17]. Intuitively, this is the bottom level for program optimization and also where the highest gains are possible [23]. Such executable and *"snap"* code, i.e., powerful for execution while free of "walled-garden", drive us to explore whether it is possible to share machine codes among COTS mobile devices over existing infrastructure, e.g., content distribution network (CDN). In Fig. 4, we further evaluate how exactly the "snap" code benefits users in contrast to either cloud app or code offloading. We observe a dramatic benefit of 19.07X speed-up relative to cloud app, also a more than 4s latency reduction and over 9 percent energy saving compared to off-loaded cases.

Guided by these findings, we provide SnapCode design to achieve the computing paradigm at OS layer.

3 Intuitions Behind SnapCode

This section presents our domain specific insights which help us achieve the aforementioned design. The first insight is that running machine codes directly can bring significant benefits. To maximize the optimization, we leverage the second insight that it is possible to share most machine codes with the same instruction set. We conclude this section by highlighting two outstanding issues in translating these insights into a practical system.

Insight 1. *Using machine codes for optimization in frequent app update behaviors.*

Today's mobile app demands frequent updates to catch up with better user experience or address potential risks for

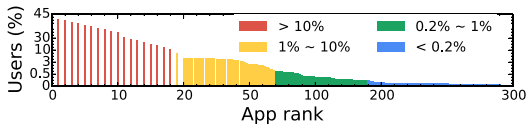


Fig. 3. Usage of traced apps within our dataset.

3. The experiment results are referred to apps' downloading, compiling and launching. While the using of app is not counted here.

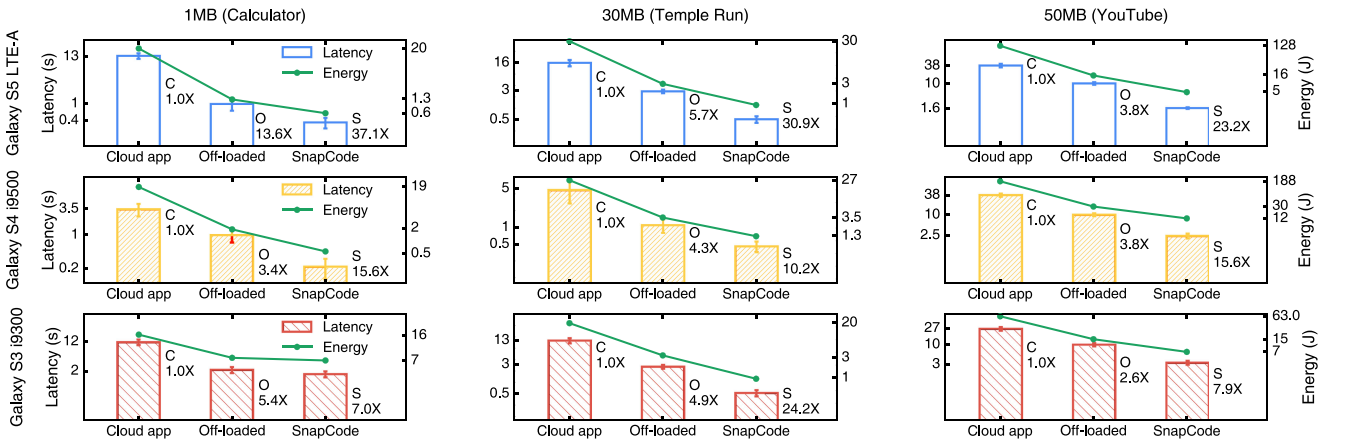


Fig. 4. Overheads of using three popular apps on Samsung Galaxy S3 (802.11n), S4 and S5 (802.11ac). Three computation mechanisms are using "cloud app" (annotated with *C*), "off-loaded" codes (annotated with *O*) and "SnapCode" (annotated with *S*). The speed-up factor is compared with native execution.

security [24], which results in heavy burden on user's data or battery life budget. For example, Chrome requests more than 70 MB in app update. In practice, SnapCode's machine code offloading could be superior in such scenarios as it respectively tackles compilation data (i.e., machine code) and static resource (e.g., images) to improve system overheads. If there are frequent app installations/upgrades, SnapCode does consume user more traffic flow. However, it only uses 520 bytes of data flow (61.7 MB as baseline) for the follow-up running, i.e., $8.04 \times 10^{-4}\%$ for regular uses. For new mobile devices, Table 2 demonstrates the superior of SnapCode comparing to default Android ART approach and existing code offloading approaches along energy cost (64 percent saving), initialization latency (82 percent decrease) and app launching time (99.4 percent decrease).

Insight 2. *Using machine codes for longer battery life.*

As Table 2 demonstrates, both time and energy overheads for app use are primarily incurred by compiling original codes into machine codes. All a device can do is to process machine language, including genetic programming [25], and all operations we conduct will be executed as machine code in the end. Thus, machine code programming is often used when there is a need for very efficient solutions, e.g., in applications with hard constraints on execution time or memory usage [26]. By now, the most efficient optimization is still done at the machine code level. The optimization could be for speed, space, or both. Genetic programming could be used to evolve short machine code subroutines with complex dependencies between registers, stack and memory. This further enables the device to get rid of any remained data.

TABLE 2
Energy/Time Cost of 5 Consecutive Uses of Chrome App on New Device

	Init.	1st use	2nd use	3rd use	4th use
SnapCode	1.30J/4.3s	0.03J/0.2s	0.03J/0.2s	0.03J/0.2s	0.04J/0.2s
Android ART	3.64J/23.9s	4.39J/23.8s	4.89J/25.9s	4.20J/27.2s	4.19J/26.3s
Thread offload	2.93J/13.0s	2.92J/12.8s	3.27J/12.7s	3.57J/13.7s	3.56J/13.8s
Method offload	4.22J/28.6s	4.98J/28.6s	3.95J/29.1s	4.67J/28.0s	4.73J/28.3s

On this basis, SnapCode intuitively seeks to enable users to boost app efficiency of their smartphones on-the-fly. In addition, it promises an energy-saving and clean experience at the same time.

Insight 3. *Sharing machine codes among Android community to maximize the benefits.*

Fig. 3 demonstrates popular apps have over 10 percent usage, i.e., more than 9,000 people own same apps. It also gives an insight that the majority can be benefited if the minority share their machine codes. In addition, as running machine codes often requires additional static resources, e.g., images and videos, to provide input-to-display components. It asks for efficient latency when aiming at a native-like experience [27], [28], [29]. Similar to [20], [30], SnapCode employs code caching strategy which generates machine codes and static resources on cloud for future sharing. Users can download them when requesting for services.

Practical issues and research questions.

There are two issues in implementing SnapCode.

RQ 1) *How to extract machine codes from today's OS?*

Indeed, it is of great engineering challenge to extract machine codes as it is strictly forbidden by modern OSes from the application layer [31]. Fortunately, Android, the open-source mobile operating system developed by Google, helps us design and implement such specialized tools in the underlying layers, e.g., userspace or OS kernel. Thus, we demonstrate our design by taking a prototype system atop Android as a running example, which integrates its source code into the AOSP project (Section 5). In principle, our reference design is also applicable to other OSes with more industrial support in the future.

RQ 2) *How to identify a compatible Android device?*

As aforementioned, it is crucial to verify the compatibility between machine codes and devices. To this end, SnapCode identifies target devices with the same runtime parameters for a comprehensive compilation. In addition, modern Android offer developers the `System.getProperty("os.arch")` interface to verify the instruction set. In practice, it usually returns `"arch64"` or `"x86_64"` to identify a 64 bit ARM/Intel architecture. In very rare cases, it either returns `"mips"` or `"mips64"` to report the MIPS

TABLE 3
Feature-Benefit Matrix for SnapCode: The ✓ Shows the Key Features of SnapCode that Contribute to Each Perceived Benefit While the × Conversely

Benefits	Features		
	Mach. (Section 4.2, Section 4.3)	*Shield (Section 4.4)*	*CDN (Section 5)*
No user data	✓		
Low latency	✓	×	✓
Energy saving	✓	×	
Less data flow	×		
More secure	×	✓	

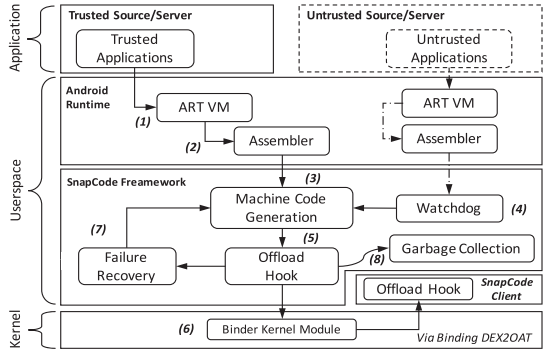


Fig. 5. Conceptual workflow of SnapCode.

case. By doing these, we are able to correctly match machine code with its compatible devices.

4 SnapCode Detailed Design

SnapCode benefits user with fast, clean and energy-saving mobile app experience. In this section, we present our design details within the scope of today's available techniques.

4.1 Overview

Table 3 summarizes the perceived benefits contributed by each feature of SnapCode at a high level. The quantitative performance benefits—low latency, energy saving, and less traffic flow— essentially arise as a result of running machine code directly and leveraging CDN infrastructure. SnapCode also has a set of qualitative features like no user data and security for better experience.

In particular, Fig. 5 depicts SnapCode's workflow which works between userspace and OS kernel. Based on customized Android (Fig. 6) with SnapCode framework, e.g., partially re-compiled AOSP project, it demands ***no*** *super-user* (i.e., ROOT) privileges (Section 5). For a small number of peers who generate and share machine code, app is loaded from a source/server with original source codes and static resources (e.g., icons) packed into an APK file. This operation then invokes a native method *(1)* that interfaces with the ART[4] interpreter and then *(2)* a linking process is called for *(3)* generating OAT file and further extracting machine codes [33]. Note that, if machine codes were untrusted, a WatchDog component would be invoked *(4)* for shielding security. Moreover, *(5)* SnapCode migrates machine code (static saved by ART) to CDN server for comprehensive security checks, thus *(6)* other devices can synchronize such executable codes for direct uses. Once failed, *(7)* SnapCode turns to a recovery step under configured limited times. Last, user is able to cache the machine codes and static resource locally for regular uses or clean them up without any user data left. Either of the decision starts *(8)* garbage collection function.

Based on our design, the whole process of code sharing from source to destination is enabled as shown in Fig. 7. Implementing this workflow and sharing process requires to address several challenges, including: *a)* machine code extraction, *b)* OS binder for sharing executable codes in streaming, *c)* secure shield for defending malicious attacks, *d)* garbage mechanism for managing user data, and *e)* recovering from failure. The remainder of this section presents our detailed design.

4. Andorid introduces ART instead of Dalvik VM since Lollipop (5.0) [32]. Thus, we primarily design SnapCode to work with ART.

4.2 Machine Code Extraction

Android runtime (ART) is the managed runtime used by applications and some system services on OS. At install time, ART compiles apps using the on-device `dex2oat` tool that accepts DEX files as input and generates a compiled app executable for the COTS device. As shown in Fig. 7, ART uses ahead-of-time (AOT) compilation, which means that, at installation, .DEX code is compiled to native code (the executable codes) in OAT files.

Algorithm 1. Machine Code Extraction Algorithm

Input: App name.
Output: Executable codes
/* Get local runtime parameters in compliation */
parameter ← getCompilingParameters();
/* Extract machines codes from OAT file */
File Oat ← getMachineCodeCache(*app*);
/* Offload to CDN via HTTPS/2 protocol */
HttpsSend(Oat, parameter, CDNserver);
/* Get machine codes with parameters */
return executable codes;

In modern Android, i.e., after Lollipop, the OAT files are mapped to memory (and are thus page-able) directly. Thus, through linking the image file contains pre-initialized classes and objects from the Android framework JARs, OAT

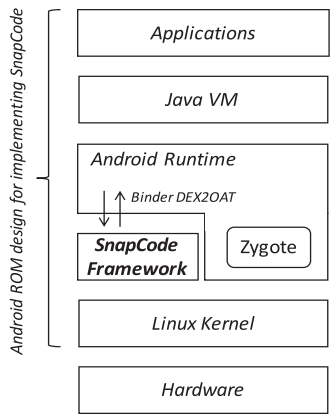


Fig. 6. Customized ROM.

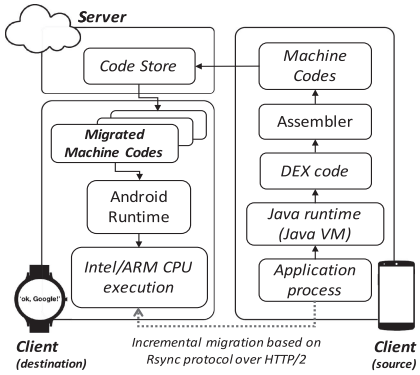


Fig. 7. Machine code sharing.

files can call methods in Android framework or access pre-initialized objects directly. In particular, Algorithm 1 illustrates the methodology of SnapCode whereby listening such process, we delegate parameter passing to the framework and then extract executable codes, i.e., machine codes, from OAT files for achieving the code sharing intention.

4.3 OS Binder for Codes Sharing

As aforementioned, SnapCode shifts code sharing from the application layer to the OS layer, where demands connection of upper apps and underlying hardware. To this end, we design the OS binder component to synchronize machine codes from CDN and incrementally delivers data flow atop Rsync protocol. In practice, when sharing machine codes from CDN (step (6) in Fig. 5), SnapCode invokes an OS binder to patch objects, i.e., machine codes, static resources and environmental parameters, to Android DEX2OAT interfaces. The following pseudocode presents the principle of our OS binder whenever calling for the machine code execution.

```
class OSBinder extend Installer {
  public void onInstallingApp(app) {
    //Delegate parameter passing to SnapCode
    Parameter parameter = get_Parameters();
    File Oat = getOatTarget(parameters);
    GetMachCode(Oat, parameters, "Rsync");
    //Bind DEX2OAT to run executable codes
    _SnapCodeBinder();
  }
}
```

Afterwards, any instruction set compatible devices, e.g., ARM- or Intel- based, can carry out the machine codes directly via calling the `_SnapCodeBinder()`. Moreover, to guarantee a secure OS binder, SnapCode uses ADBI toolkit [34] to hook the functions of `CallNonVirtual<Type>Method` family. By doing this, all calls to these functions are checked by SnapCode to block calls to an hooked virtual-method if these calls do not come from SnapCode framework.

Update Process with Rsync. In our current implementation, we synchronize each pair of versions with rsync protocol for local cached machine codes. To this end, SnapCode sends the "COPY" commands to the target in topologically sorted order. Upon sending a COPY command, it deallocates the corresponding repository along with its remaining edges. After sending all COPY commands, SnapCode sends "ADD" commands according to the add list, including the ADD commands that correspond to deleted COPY commands. If necessary, the target truncates the machine code files to the new size and the machine code update synchronization is complete.

4.4 Security Shield

Running the migrated machine code directly can raise concerns about system security. To provide adequate protection for users as well as keeping the efficiency, we take all potential vulnerabilities into consideration for the *Watchdog* module design, including: (1) Untrusted server: an untrusted server could take arbitrary action (e.g., altering and replacing) on the migrated machine code; (2) Malicious clients: a malicious client can upload altered/arbitrary machine code to the server, which greatly harm other valid users. (3) Insecure communication channel: machine code transmission over the networks especially the wireless networks is highly vulnerable to network security threats. (4) Untrusted third-party applications: among billions of applications in the market, many of them are mal-wares, which could cause system crash or data leakage. Our framework leverages a whole set of light-weight mechanisms to deal with these vulnerabilities.

First of all, a valid SnapCode server is authenticated by a certificate, which is authorized by a trusted "certificate of authority" and can be examined by any client. We assume the authenticated server is trustworthy and will conduct the legitimate operations faithfully. Each APK has its unique digest which is recorded by the server. Then, we need to make sure all clients execute valid SnapCode application, all generated and uploaded machine codes are correct. To validate a client, when a user installs the SnapCode client from an authorized server, she will be provided a unique pair of keys (a public key and a private key) associated to her account and device identity using an highly efficient signature scheme, e.g., HORS [35], [36]. Before uploading a piece of machine code produced by a SnapCode client, a digest of this code will also be generated and signed with the client's private key. Then the code as well as the signature will be uploaded to an authorized server. The server maintains detailed tables of valid users and applications, and exams the uploaded signature first and then the digest of the machine code to guarantee the validity of the client and correctness of the machine code (i.e., the digest is consistent with those of the same application from other valid users). A pair of digests of an APK and its correct machine code is recorded by the server. In this way, untrusted clients who have uploaded incorrect machine codes can also be detected and tracked by their private keys, and the incorrect codes and all subsequent codes from untrusted clients will be abandoned. We adopt the HTTPS/2 encryption protocol to ensure the integrity and confidentiality of all the traffic of SnapCode, which is resistant to eavesdropping, tampering and man-in-the-middle attacks. Integrating encrypted communication with access control mechanisms [37], [38], the user data uploaded to the cloud can be protected from unauthorized parties. To further improve the whole system security and user experience, we employ the untrusted third-party application analysis methods as a building block of our system, such as DroidScope

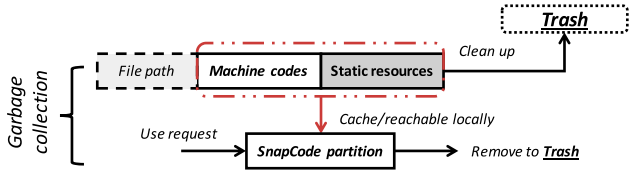


Fig. 8. Process of machine codes and static resource caching locally or cleaning to the Trash.

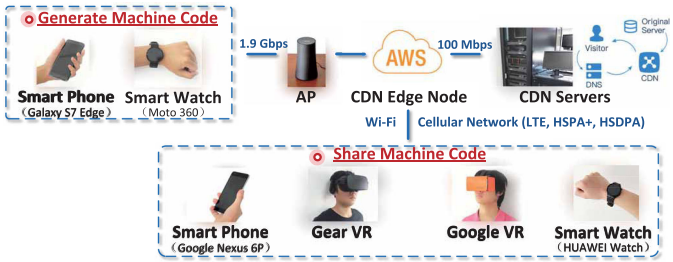


Fig. 9. Android-based testbed for SnapCode framework.

[39], TaintDroid [40], and FlowDroid [41], to identify and label misbehaving applications. Since, the application analysis could be burdensome, it will be executed on the server in an asynchronized manner. For each version of each application, the detection only need to be executed once for all. The server will mark both digests of each APK and its machine code with the detection result.

For other common system security problems, e.g., DDOS attack [42] and Chosen-ciphertext attack, SnapCode leverages existing proper schemes to prevent them since they are not the focus of this study.

4.5 Garbage Collection

The executable codes and static resources as described so far will keep growing indefinitely. Intuitively, some of the shared apps (in machine codes) will no longer be needed if a user wants a clean experience and it can be removed. To resolve this issue we design a garbage collection mechanism. It can be either manually triggered or automatically invoked when a normal garbage collection has failed to free enough memory and the ART is about to signal it is out of memory.

To conduct the garbage collection, as Fig. 8 demonstrates, for a specific app, SnapCode marks every object reachable locally for addressing, i.e., file path, then terminates all processes of this app from ART. Once done, the app's executable codes and static resources would be moved to the *Trash* temporally. Note that such design can benefit user of either OS or user-related data free. In addition, objects in the Trash can be recovered in case there is an accidental gesturing. We shall emphasis that objects in Trash would be cleaned up automatically if no further access during a period. On the other hand, such objects can also be cached locally in SnapCode's own partition for regular uses until been removed as aforementioned.

4.6 Failure Recovery

From the design of SnapCode, failure recovery comes almost for free. To properly implement failure recovery, i.e., the client can enter a state that it could recover from, every single operation that can be executed must either be suspended for all remote acknowledgements to arrive before removing any machine codes (as data is buffered locally); or be killed before a regular synchronization completes. To recover from a failure the client needs only resume all threads locally and reset the machine code synchronization.

Specifically, in the case of resuming data downloading from CDN, for each synchronization the server is required to send an acknowledgement of all thread stacks. In this way, once connection lost, the server has the necessary stack information to resume execution. Reversely, the client, however, needs only to check whether the migrated machine codes have been cached on the server. In addition, the detection of server loss is as simple as the HTTPS/2 connection to the server is closed or the server has not responded to a heartbeat soon enough.

5 Implementation and Deployment

This section presents our implementation of SnapCode framework and highlights engineering solutions to address practical challenges (e.g., avoiding Android ROOT permissions and speeding up deployment with CDN).

Android Prototype Implementation. We have implemented SnapCode prototype framework on the basis of Android 5.0+ with our customized ROM, which requires no further *super-user* (i.e., ROOT) permission from users. Specifically, in this paper, we explore SnapCode by running it over heterogeneous COTS Android devices, i.e., Samsung Galaxy S5 (LTE-A), S4 (i9500), S3 (i9300), LG Nexus 5 and Moto Nexus 6, HUAWEI Nexus 6P smartphones; Moto 360 watch and HUAWEI smartwatch.

In Android ecosystems, e.g., Samsung, LG and HUAWEI, such OSes and underlying libraries are not fully accessable without the privilege of ROOT, which, however, often leads to vulnerable security or privacy issues. To avoid such problem and protect the entire OS from malicious attacks, by partially re-building the AOSP [18] codes, we integrate SnapCode framework into the original ROM (as aforementioned in Fig. 6), and employ such a ROM to the experimental devices.

Since running SnapCode framework requires client to synchronize machine codes from the CDN server, meanwhile it incurs additional traffic flow for loading input-to-use static resource, e.g., images and audios, it raises critical demand for a low and stable service latency. Considering today's commercial network and prior studies such as [3], [4], we adopt the 802.11, e.g., b/g/n (2.4 GHz) or ac (5 GHz), LTE and 3G communication techniques in our prototype, as presented in Section 6.3.

Network Testbed Deployment. To demonstrate the feasibility of SnapCode within the scope of today's available techniques, we deployed a CDN-based testbed (Fig. 9) to carry out the relevant experiments. The testbed was composed of Amazon Inc's cloud server (using AWS t2.micro case), self-built CDN servers, campus WiFi and commercial cellular network (FDD-LTE and HSPA+). Table 4 summarizes the detailed configuration and link speed of our testbed. Specifically, there are two categories of cloud servers providing SnapCode networked services, i.e., *1)* AWS-based CDN edge node for redirecting the service request, and *2)* CDN servers for caching extracted machine codes as well as

TABLE 4
Testbed Setup for Benchmark Evaluations

Local area network (LAN)		Wide area network (WAN)	
Access pattern	Uplink/downlink	Access pattern	Uplink/downlink
802.11ac	0.82/29.30 MBps	FDD-LTE	0.35/9.40 MBps
802.11bgn	0.54/12.87 MBps	HSPA+	0.08/2.08 MBps
AWS EC2 (t2.micro) configuration			
1 vCPU, 2.5 GHz, Intel Xeon Family, 1 GiB Memory, 100 Mbps bandwidth			

TABLE 5
Information of Participants in Our Case Study

Access pattern	Avg. usage/day	Avg. data budget
WiFi	11.8 hours	1.36 GB/month
Cellular	6.3 hours	

detecting malicious objects with the *WatchDog* component we designed in Section 4.4. Moreover, we carry out experiments with any mobile devices that generate and migrate machine codes to the cloud, and then other devices share the available codes via HTTPS/2 and Rsync protocols.

Power Meter Setup. As our goal is to benefit on-the-go users with clean (no user data) and high efficiency experience but charge little cost, the energy consumption metric is extremely significant. However, today's mobile devices often provide no way to obtain fine-grained energy measurements of any apps [43]. Instead, they simply present users a message like *"how much battery (in %) is left"*, which is very coarse-grained and usually unreliable [44]. In this paper, to precisely measure devices' energy cost, we attach a hardware power meter [45] to Samsung Galaxy S3, S4 and S5 smartphones with hijacked batteries (Fig. 10) that provide the same performance as the normal battery to ensure valid evaluations in our work. Note that this power meter provides fine-grained measurements: it samples the current drawn from the battery with a frequency of 5 KHz and a mean error of less than 3 percent [43], [45].

6 Evaluation

This section evaluates the performance of SnapCode and its benefits. Specifically, to conduct comprehensively exploration, we begin by breaking down the potential benefits according to Section 4.1. The first class of qualitative but subjective features are measured by user studies (Section 6.1), i.e., experience rating, whose results are need to be coupled with on-the-go users. The second class of quantitative and objective features stem from the ability to optimize latency and overheads for app compilation, i.e., installation or upgrade (Section 6.2) as well as regular use (Section 6.3), measured by trace-driven experiments against benchmarks.

6.1 Case Study

We first investigate the qualitative benefits of SnapCode with real-life uses.

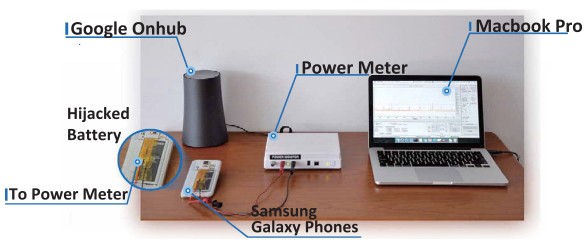


Fig. 10. The hardware power meter used for energy measurements with Android smartphone.

Study Design and Instrument. To better understand how machine codes sharing promotes users' daily experience, during August 10th to 17th 2016 and within the range of our campus, we conducted a user study in which 40 volunteer participants (12 females and 28 males) were asked to comprehensively experience (install/update and regular use) the most popular 20 Android apps from Google Play (Table 7) including *Chrome*, *Uber*, *Maps* and *YouTube*. Each participant was invited to use those apps provided by SnapCode method and default method for at least 15 minutes respectively. For this study, two Samsung Galaxy S5 LTE-A smartphones had been flashed with the aforementioned customized ROM (Section 5). The first S5 smartphone loaded apps using SnapCode solution (without caching codes), the other employed default Android app mechanism (i.e., fetching from app market) as baseline. Both devices were connected to either 802.11n/ac campus WiFi or LTE. We asked all participants to complete a survey after using both two solutions. In all user trials, we capture each of app response time, size of remaining files on the device,[5] and battery life reduction to validate the effectiveness of their evaluations. The survey contained a consent form, and asked for the following information: basic demographic information, their concerns for daily mobile app use, as well as their monthly data budgets and daily WiFi uses condition, which is summarized in Table 5 to further validate the commercial feasibility of SnapCode.

Specifically, participants were asked to rate their experience, given five rating levels, i.e., one ("Poor"), two ("Fair"), three ("Good"), four ("Excellent"), and five ("Perfect") according to questions like: *"I consider it has a faster response (launch quickly) experience when I install/update/use apps?"*, *"I consider it has fewer energy cost after a long time use?"* and *"I think it won't leave any user data on my smartphone?"* etc. Furthermore, we used these ratings to evaluate how SnapCode benefits users with qualitative experience improvement.

Result and Analysis. Of the 40 participants, 18 are with 0.82/29.30 Mbps (802.11ac) uplink/downlink speed, 12 are with 0.54/12.87 Mbps (802.11n), while the rest are with 0.35/9.40 Mbps (FDD-LTE). Based on our survey, the average cellular data budget per month is 1.36 GB while over 70 percent users would be WiFi available for at least 11.8 hours everyday. Note that, in the heaviest use, people spend no more than 5.6 hours on using mobile devices [46], which indicates that SnapCode can significantly improve their experience with little cellular flow cost in most cases. In addition, our statistics (Table 6) show that all participants voted for an excellent or perfect performance in speeding up app with SnapCode. More than 87.5 and 92.3 percent participants evaluated SnapCode to be excellent in saving

5. Participants are introduced to look at the storage usage of apps for evaluating the local data usage.

TABLE 6
Comparison between Mean Opinion Scores and Values of SnapCode Against Phone-Alone Solution

Mean Opinion Score	Speedup	Energy saving	Clean
1 (Poor)	-	-	3.85%
2 (Fair)	-	10%	3.85%
3 (Good)	-	2.5%	42.31%
4 (Excellent)	40%	52.5%	30.77%
5 (Perfect)	60%	35%	19.23%

their battery life cost and good enough in achieving no data remaining after uses, comparing to the default Android approach, respectively.

In summary, our user study indicates that SnapCode can benefit users with perceivable faster and cleaner experience with little energy cost, which demonstrates the significance of our system.

Public Deployment. We deployed SnapCode testbed with smartphones and smartwatches publicly on Tsinghua and CSU campuses. Around 500 participants used our system and we have received many oral ranking from them where more than four in five reviews are positive.

6.2 Macrobenchmarks

Our case study demonstrates that SnapCode's machine codes sharing solution can provide significant improvement of use experience. In this section, we measure our Android prototype in a trace-driven manner to quantify optimization of latency, energy cost and throughput for app installation/upgrade where compilation on SoC makes significant sense [47]. For every single experiment, at least 100 tests are conducted with persistent process. Furthermore, we consider the state-of-the-art work as marcobenchmarks, and 20 most popular apps on Google Play (Table 7) are adopted as input metrics. The benchmarks and our solution are enumerated as follows.

- Cloud app (CA) solution natively runs app from a cloud store, which work as Chrome OS [48] and Google web store [49].
- Process offload (PO) approach offloads app's every threads to a cloud VM (i.e., the thread code offloading) to achieve the CloneCloud-like [3] effect.

TABLE 7
Traffic Flows (MB) and Speedup Factors of 24 Popular Apps Powered by SnapCode

App	Size	Speedup	App	Size	Speedup
WPS Office	111	13.98x	Skype	64	9.86x
Uber	69	12.34x	Wechat	103	9.52x
Instagram	35	11.76x	Dropbox	62	9.11x
Tumblr	65	11.14x	Maps	104	8.92x
Facebook	236	10.94x	Pokémon Go	72	8.90x
Twitter	75	10.77x	Spotify Music	57	8.74x
YouTube	51	10.70x	NBA Live	71	8.08x
Amazon	68	10.63x	Snapchat	106	7.58x
WhatsApp	45	10.57x	Chrome	74	7.37x
Gmail	35	10.26x	Clash Royale	102	6.79x
Need for Speed	696	25.43x	NBA 2K16	1890	10.56x
Final Fantasy	1815	17.09x	FIFA 16	1123	16.97x

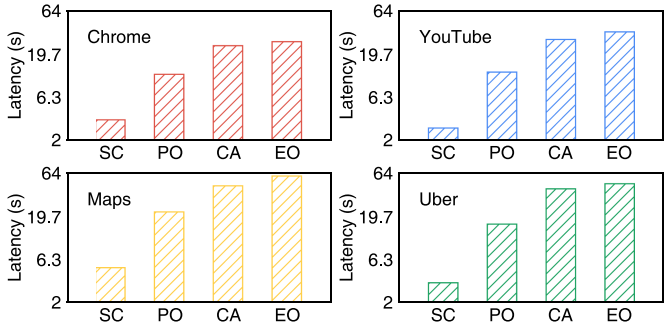


Fig. 11. Comparison of latency with four most popular apps as the input.

- Execution offload (EO) approach offloads app's every method or class to achieve a COMET-like [4] execution migration (i.e., the method/class code offloading).
- SnapCode (SC) solution uses SnapCode (i.e., the machine code offloading) to run machine codes directly and bypass any preliminary steps.

Speeding Up. We first measure the latency by using 4 popular apps as input metrics to the benchmarks on Samsung S5 LTE-A phone under 802.11ac environment. The geometric mean results are shown in Fig. 11 and annotated with numerical factors which indicate the relative latency improvement (speed-up factors) compared to CA solution. We see a significant speed-up powered by SnapCode for all four metrics. Specifically, SnapCode averagely benefits users with more than 24 s latency reduction in contrast to the cloud app solution. In addition, there appears a performance gap among different metrics (apps) within a specific benchmark. For instance, SnapCode can speed up Uber by a factor of 12.3x while only 7.3x for Google Chrome. We reveal the measurement results for 20 most popular apps in Table 7. By running shared machine codes on smartphone, SnapCode can speed up an app's (around 110 MB) installation (or upgrade) with an averaged factor of *9.90X*, a peak factor of 13.98x and a valley of 6.79x.[6] Note that, the latencies when using SnapCode are usually less than 5.1 s, while the same operations cost users more than 59.2 s through CA solution. As benchmarks, the PO approach provides users with only 2x speed-up, while the EO approach increases the processing time when using low precision computational apps, e.g., YouTube. The result shows that SnapCode solution significantly outperforms existing solutions in efficiency. We shall further present that, by using Rsync technique, SnapCode can bring more than 120x speed-up for most apps' regular use as demonstrated in Table 2.

Energy Saving. As aforementioned, the major energy overheads in app trial lies in compiling source codes into machine codes. To explore the energy saving by using SnapCode which avoids the compilation steps for optimization, we employ above four apps as metrics again and observe that more fine-grained computation would be significantly more energy-consuming (Fig. 12). For instance, to install Chrome, SnapCode costs user a balanced energy of 0.94 J, which achieve 82 percent energy reduction of CA usage (as baseline). Meanwhile, it climbs to 1.77 J (85 percent reduction of baseline) for installing Google Maps. Upon

6. Note that most apps' traffic flow from Google Play for the compiling process are less than 110 MB, as we explore on Table 9 in Section 7.

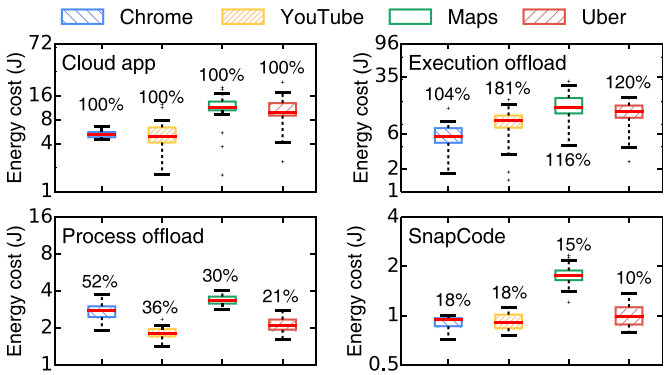


Fig. 12. Comparison of energy cost with four most popular apps as the input baseline.

comparison, the PO approach respectively consumes user an energy of 2.8 J (48 percent reduction of baseline) and 3.4 J (70 percent reduction of baseline) due to its additional compilation. EO approach is even more energy-consuming as it costs 5.5 J and 13.3 J (4 and 16 percent increase than baseline) on Samsung S5 LTE-A smartphone. The result demonstrates that our work achieve significant energy cost reduction. Moreover, with better hardware (SoC and link speed) support, SnapCode can further reduce over 80 percent energy cost and achieve a greater speed-up factor for app installation, which has been illustrated in Fig. 4 compared to the performance of Samsung S3 and S4 smartphones. By compressing static resources such as icons, the performance of SnapCode can still be improved.

Throughput Requirement. We then measure the adaptability of SnapCode to dynamic channel conditions by installing and using Chrome app under 802.11ac environment. For app's compilation, Fig. 13 measures that the peak throughput of SnapCode reaches 28.5 Mbps. In most cases, SnapCode consumes 22.1 Mbps on average. For comparison, CA (EO, PO) requires a balanced throughput of 21.6 Mbps (16.6 Mbps, 22.3 Mbps). During compilation, SnapCode consumes more traffic flow than CA, while EO requires fewest traffic. This is because SnapCode loads additional static resources than CA. In contrast, EO only offloads the computation data to cloud and holds on to execution with additional time consumption. However, for regular uses, SnapCode demands far less throughput than other approaches as shown in Fig. 14, where the throughput cost of SnapCode reduces dramatically since the second use of app.

6.3 Mircobenchmarks

The above evaluations demonstrate the superior performance of SnapCode on new devices. This section looks at mircobenchmarks that conduct user actions of sharing machine codes. Specifically, we measure how SnapCode

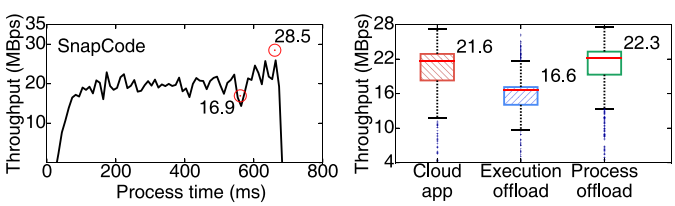


Fig. 13. Throughput meter to Chrome compilation.

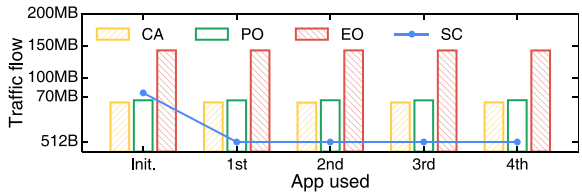


Fig. 14. Traffic flow of the first 5 uses of Chrome.

performs with comparing to default Android ART mechanism, i.e., using reusable objects.

How Much Energy Needs to be Cost?. We expect to benefit the efficiency of energy if users would like to share (download and cache) machine codes and then use it with their mobile devices. Specifically, we measure the energy cost of each downloading operation (by averaging at least 100 standard usages) in according to different data sizes.[7] Fig. 15 shows the negligible energy consumption for machine code downloading from 1 MB traffic flow to 230 MB at varying intervals under 4 network conditions. We observe that, taking the 802.11ac and FDD-LTE cases for instance, SnapCode effectively consumes 29 mJ ($7 \times 10^{-7}\%$ battery life) and 43 mJ ($1 \times 10^{-6}\%$ battery life) for each machine code sharing, and the energy cost variation are all less than 6.2 mJ. Comparing to the default Android ART mechanism (compiling reusable objects), our solution can benefit more than 90 percent energy saving in all scenarios, which indicates the significant benefits from SnapCode.

What are the Implications with SnapCode?. As SnapCode runs machine codes directly rather than compiling the reusable objects before use, it can dramatically reduce apps' launching time. To shape the exactly implications of SnapCode, we measure the launching time of 20 most popular apps. Fig. 16 depicts the cumulative probability distribution (CDF) of launching time for app uses on S5 LTE-A smartphone. We find that SnapCode with 802.11ac access achieves the best performance (180 ms on average) in responding to user request. In the 802.11bgn and FDD-LTE cases, both launching time are 185 ms on average, which almost tie to the best case. In the HSPA+ case, the cost is 255 ms on average. Considering real-life experiences [50] and our experimental results in Table 2, we summarize that SnapCode achieves good efficiency and provides user as well experience as default Android ART (0.1 s).

7 Discussions

This section discusses the practical issues and ongoing work of our current implementation. Broadly, there comes out to more data flow demands of security shield penalty.

Security Shield Penalty and Improvement. Security shield defends users against untrusted parties in insecure environments. To minimize the penalty, our design keeps security shield light-weighted and outsources the burdensome work to the powerful server. The server is responsible for client validation, personal data management, uploaded code correctness verification and untrusted application analysis. For

7. Note that for a specific traffic flow, there might be multiple apps. E.g., both Dropbox and AntiVirus are 61.7 MB as shown in Table 9 in Section 7.

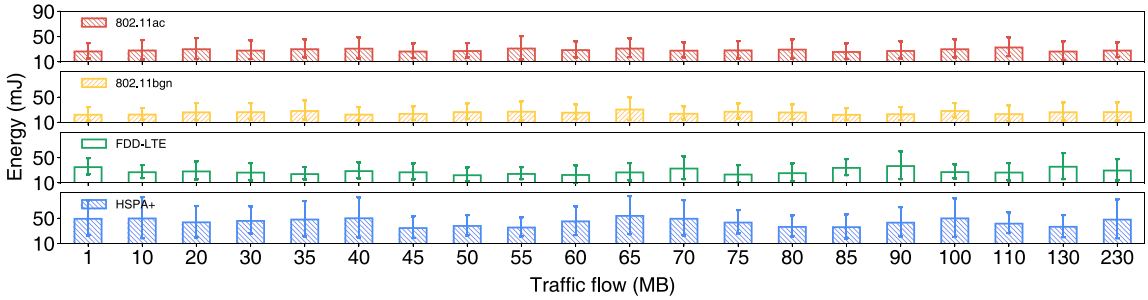


Fig. 15. Energy meter to different traffic flow increase as the input baseline for measurement.

each version of every single app, the analysis only need to be executed once for all and more advanced schemes can be adopted with the development of machine code analysis techniques. All these computation is carried out before app usage, thus will not affect the user experience. To measure the perceptible experience penalty caused by authentication and encryption, we evaluate the energy cost and latency with enabling secure function. As Fig. 17 shows, SnapCode consumes diverse energy costs under different networking conditions (802.11ac, 802.11bgn, FDD-LTE and HSPA+), while the extra cost for security protection is only about 10 percent. The latency penalty is below 0.6 s in all cases, as presented in Fig. 18. Both figures show that, introducing security shield provides users a more trustworthy system with only a small price, which still keeps SnapCode far more energy-saving and efficient than traditional methods. Note that, all security methods, e.g., encrypted communication, access control and untrusted app analysis, are adopted as building blocks of our design, and can be replaced by more advanced methods as the development of system security study.

Ethics. Throughout this project, we took the utmost care to protect users and their sensitive data. Users have total freedom to choose to install SnapCode. Specifically, our users explicitly opt-in to data collection. The opt-in screen clearly informs users what data will be collected, and users are free to opt-out at any time in the settings (or by uninstalling). Furthermore, the dataset used in our experiments was securely stored on the cloud servers, and at no time did user data leave our system.

RTT Responsiveness from Sink to Source. As Fig. 16 has demonstrated that higher network RTT can impact SnapCode efficiency tremendously. We carry out RTT measurements and summarize the results in Table 8. In practice, many of the apps used in our experiments, e.g., Facebook, Google, etc., would have access issues in China. To avoid such issues and ensure that any perceived gains are from SnapCode, we employ to use socks5 proxy to moving the data from content provider (i.e., the source) to us (i.e., the sink). Specifically, we deploy our proxy server (i.e., the transport) our test server in Tokyo, Japan. We find that the RTT value from the transport to the source is about 1.35 ms, which is very negligible. Furthermore, the total RTT from the sink to the source are all less than 190 ms under any network environments, which can bring efficient and practical quality of service for most of mobile users [51].

Performance Analysis of SnapCodeon Smartwatches. To understand latency benefits of SnapCode on smartwatches, we conduct performance analysis on the client side with four popular, diverse user apps: Chrome, YouTube, Uber and Maps. In each case, we keep the app for five minutes use with and without SnapCode enabled while connected to a WiFi network.

For latency test in SnapCode, Figs. 19 and 20 directly record the service latency for evaluating each test. Note that either Moto 360 or HUAWEI watch supports 802.11b/g

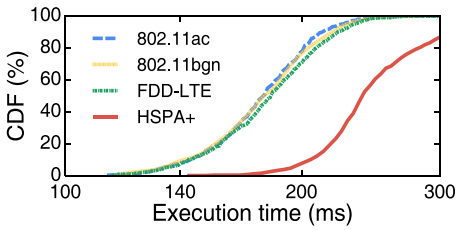


Fig. 16. Execution time (CDF) of 20 popular apps.

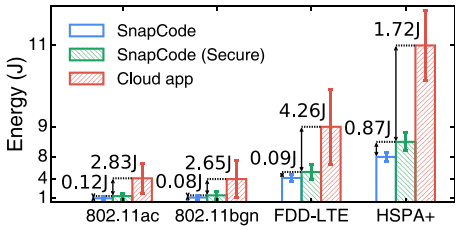


Fig. 17. Energy cost with enabling secure function and cloud approach.

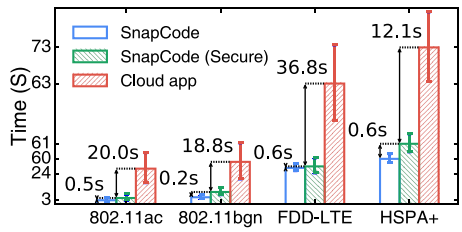


Fig. 18. Latency with enabling secure function and cloud approach.

TABLE 8
RTT Responsiveness from Sink (Local Device) to Transport (VPN Server) to Source (Google Play Server)

Access pattern	802.11ac	802.11bgn	FDD-LTE	HSPA+
Sink to transport	87 ms	99 ms	145 ms	182 ms
Transport to source	1.35 ms	1.35 ms	1.35 ms	1.35 ms
Sink to source	89 ms	101 ms	147 ms	184 ms

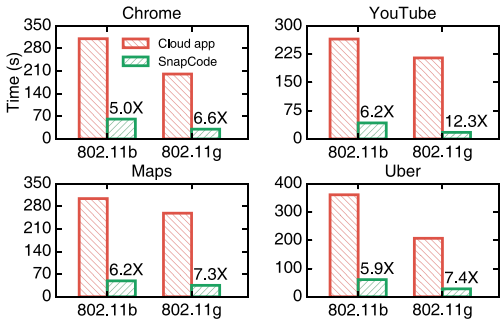


Fig. 19. Speed-ups to different apps on Moto 360 watch.

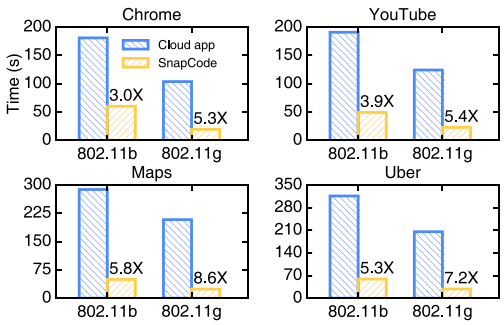


Fig. 20. Speed-ups to different apps on HUAWEI watch.

only, which can yield up to 54 Mbps downlink bandwidth in theory [52]. In practice, we get the machine codes from our deployed testbed (Section 5). For the four apps, we see more powerful SoC (HUAWEI watch) and network condition (802.11g) benefits user fewer latency. Specifically, it comes out to 8.6X speed-up factor when in using Maps on HUAWEI smartwatch, which, in fact, reduce user around 200 seconds in regular use. It again demonstrates the effectiveness and significance of SnapCode design.

Performance Analysis of SnapCodewith Mobile VR. We also investigate how SnapCode benefit mobile VR apps in Fig. 21. By comparing Samsung devices (Gear VR) and Google devices (Google Cardboard VR), we observe that SnapCode speeds up user significantly with average 6.4X factor.

In detail, we plot the distribution of all the latency reduction ratios across network condition. We observe that, in original method, Gear VR has a faster performance than

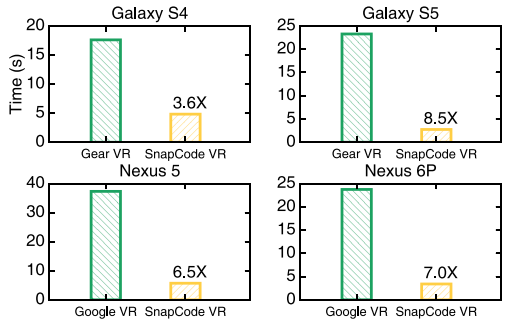


Fig. 21. Speed-ups to different VR apps on smartphones.

Google VR, which reaches around 8 s. In comparison, SnapCode provides user no significant differences wherever running on any devices. We highlight the stable performance of SnapCode and will extend it to work with either more OS, e.g., iOS, or wireless devices, e.g., smart-home.

How Many Data Needs to be Transferred When Using SnapCode?. We also explore SnapCode's data flow consumption for installing/updating the most popular 100 apps on Google Play. Specifically, Table 9 summarizes their information which ordered by greatest popular reduction. We observe, when linked native libraries and static resources, the data flow can be larger than install package, i.e., APK file. For instance, the APK file of Fackbook is 99 MB, while it reaches 235.9 MB in using SnapCode. Thus, we treat SnapCode as achieving the promised benefits at a cost of more data flow. Nevertheless, it is acceptable in WiFi environment or user has sufficient data budget.

8 Related Work

SnapCode is built upon previous research done in Mobile cloud computing, Optimization by code offloading and Android optimization.

Mobile Cloud Computing. The term of *mobile cloud computing* was first introduced around a decade ago. It has been attracting the attentions of mobile users as a new technology to achieve rich experience of a variety of mobile services at very low cost, and of researchers as a promising solution [53]. Aepona Inc. describes mobile cloud computing as a new paradigm for mobile applications whereby the data processing and storage are moved from the mobile device to powerful

TABLE 9
Machine Code Sizes (MB) of Top-50 Apps on Google Play, Ordered by Greatest Popular Reduction

App	Size	App	Size	App	Size	App	Size
Facebook	235.9	Chrome Browser	73.6	YouTube	50.6	Messenger	128.2
Google Play services	126.9	Google	85.8	WhatsApp	44.8	Instagram	34.9
Snapchat	106.3	Gmail	35.2	Pokémon GO	71.9	Clean Master	48.5
Maps	104.2	360 Security	39.3	Yahoo Mail	41.9	Pandora Radio	45
Kik	66.2	Verizon Messages	88	Twitter	74.8	GO SMS Pro	48.2
AntiVirus	61.7	Uber	68.7	HTC Gallery	75.6	Clash of Clans	75.2
Spotify Music	56.8	imo	16.84	Google+	61.2	Viber	66.6
metroZONE	18.62	DU Battery Saver	23.23	Nova Launcher	15.11	DU Speed	32.4
Netflix	30.6	Pinterest	55.2	GO Speed	21.41	Power Battery	11.36
Clock	11.64	Google Photos	56.1	Power Clean	10.86	Amazon	67.6
CallApp & Block	62.3	LINE	88.9	NBA Live Mobile	71	Firefox	51.5
Skype	63.8	Tumblr	64.8	Clash Royale	102.3	Microsoft Outlook	44.9
Dropbox	61.7	WPS Office	111.1	Calculator	1.01	Temple Run	28.24

and centralized computing platforms located in clouds [54]. In addition, mobile cloud computing now has been defined as a combination of mobile web [55] and cloud computing [56], which is the most popular tool for mobile users to access applications and services on the Internet. Motivated by this insights, we design a cloud compilation mechanism for SnapCode.

Optimization by Code Offloading. A significant amount of related work has gone into optimizing mobile systems that combine the computation efforts of either multiple machines or devices. The biggest category of such works are those that use code offloading to improve the system performance. Specifically, how to overcome the resource constraints of mobile devices by partitioning programs between a device and more powerful server infrastructure is challenging. Chroma [57] provides programmers to specify program partitions and conditions under which to adopt those partitions, while MAUI [43], CloneCloud [3] and EECOF [58] leverage features of managed-language runtimes to automatically partition a program. Inspired by such studies, SnapCode looks forward a machine code sharing mechanism for further improving on-the-go mobile user's experience.

Android Optimization. Prior researches in the field of mobile phone-alone energy optimization such as backlight brightness adjustment [59]. The past six years have witnessed significant activity in optimizing Android OS, and several tools have been developed to help application developers test their designs for energy efficiency such as PowerTutor [44], MobiPerf [60] and Mobilyzer [61] etc. Several optimizations, e.g., [39], [62], [63], have been proposed at the Dalvik Virtual Machine (DVM) level, which is the software responsible for the execution, management and security of apps in the Android platform. Close to our work, Smali [64] and Androguard [65] are designed to reverse engineer Dalvik bytecodes. However, there is yet no work before SnapCode that makes mobile devices focuses on machine code sharing which bypasses the compilation for a further optimization.

9 Conclusion and Future Work

In this paper, we have proposed SnapCode, a framework aimed at running mobile app with machine code sharing from peers transparently. We introduced a machine code generation technique and OS binder operation to run our system within the scope of today's available techniques. This makes all generated code offloadable and allows multiple mobile devices to use directly. We demonstrated this system on 20 real apps on Google Play and showed an average speed-up of 9.90x. In regular uses, we were able to reach as much as 120x speed-up on average. To broaden the impact of our work, we have made the SnapCode testbed available on two universities.

SnapCode promises a new paradigm which significantly change the way we use apps in the future. In our design, proper light-weight mechanisms are also adopted to enforce security. Moving forward, facing potential emerging threats to shared machine code, the most promising line of work is in improving the secure shield (WatchDog) in the cloud server.

Acknowledgments

We thank the anonymous reviewers for their valuable and insightful comments. This work is supported by National Natural Science Foundation of China under Grants No. 61802218.

References

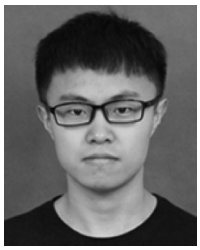
[1] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Softw.*, vol. 32, no. 3, pp. 70–77, May/Jun. 2015.
[2] "Nextbit: Robin. The smarter smartphone," 2017. [Online]. Available: https://www.nextbit.com
[3] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 301–314.
[4] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 93–106.
[5] D. C. Chu and M. Humphrey, "Mobile ogsi. net: Grid computing on mobile devices," in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, 2004, pp. 182–191.
[6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic Programming: An Introduction*. San Mateo, CA, USA: Morgan Kaufmann, 1998.
[7] R. I. Strandh, "Classes of equational programs that compile into efficient machine code," in *Proc. Int. Conf. Rewriting Techn. Appl.*, 1989, pp. 449–461.
[8] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-cc multiprocessor machine code," *ACM SIGPLAN Notices*, vol. 44, pp. 379–391, 2009.
[9] H. Yang, H. Luo, F. Ye, S. Lu, and L. Zhang, "Security in mobile ad hoc networks: Challenges and solutions," *IEEE Wireless Commun.*, vol. 11, no. 1, pp. 38–47, Feb. 2004.
[10] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annu. IEEE Int. Conf. Local Comput. Netw.*, 2004, pp. 455–462.
[11] L. Pu, L. Jiao, X. Chen, L. Wang, Q. Xie, and J. Xu, "Online resource allocation, content placement and request routing for cost-efficient edge caching in cloud radio access networks," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 8, pp. 1751–1767, Aug. 2018.
[12] A. N. Khan, M. M. Kiah, S. U. Khan, and S. A. Madani, "Towards secure mobile cloud computing: A survey," *Future Generation Comput. Syst.*, vol. 29, no. 5, pp. 1278–1299, 2013.
[13] P. K. Tysowski, "Methods and apparatus for use in transferring user data between two different mobile communication devices using a removable memory card," U.S Patent 8,233,895, Jul. 31 2012.
[14] E. Pitoura and G. Samaras, *Data Management for Mobile Computing*, vol. 10, Berlin, Germany: Springer, 2012.
[15] D. Zhang, L. Tan, J. Ren, M. K. Awad, S. Zhang, Y. Zhang, and P.-J. Wan, "Near-optimal and truthful online auction for computation offloading in green edge-computing systems," *IEEE Trans. Mobile Comput.*, p. 1, 2019, doi: 10.1109/TMC.2019.2901474.
[16] "Antutu Report: TOP 10 Global Popular Smart phone and User Preferences,1H 2016," 2017. http://www.antutu.com/en/view.shtml?id=8258
[17] "The mobile ecosystem runs on ARM," 2016. [Online]. Available: https://www.arm.com/markets/mobile
[18] "Android open source project," 2017. [Online]. Available: https://source.android.com
[19] "DU antivirus security," 2017. [Online]. Available: https://play.google.com/store/apps/details?id=com.duapps.antivirus
[20] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *Proc. IEEE INFOCOM*, 2012, pp. 945–953.
[21] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi, "Kahawai: High-quality mobile gaming using GPU offload," in *Proc. 13th Annu. Int. Conf. Mobile Syst. Appl. Serv.*, 2015, pp. 121–135.

[22] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey," *Future Generation Comput. Syst.*, vol. 29, no. 1, pp. 84–106, 2013.
[23] D. Johansen, R. Van Renesse, and F. B. Schneider, "Operating system support for mobile agents," in *Proc. 5th Workshop Hot Topics Operating Syst.*, 1995, pp. 42–45.
[24] A. Möller, F. Michahelles, S. Diewald, L. Roalter, and M. Kranz, "Update behavior in app markets and security implications: A case study in google play," in *Proc. 3rd Int. Workshop Res. Large*, 2012, pp. 3–6.
[25] P. Nordin, W. Banzhaf, and F. D. Francone, "12 efficient evolution of machine code for cisc architectures using instruction blocks and homologous crossover," *Advances Genetic Program.*, vol. 3, 1999, Art. no. 275.
[26] P. Nordin, *Evolutionary Program Induction of Binary Machine Code and its Applications*. Münster, Germany: Krehl Munster, 1997.
[27] R. Krishnan, H. V. Madhyastha, S. Srinivasan, S. Jain, A. Krishnamurthy, T. Anderson, and J. Gao, "Moving beyond end-to-end path information to optimize CDN performance," in *Proc. 9th ACM SIGCOMM Conf. Internet Meas. Conf.*, 2009, pp. 190–201.
[28] C. Huang, A. Wang, J. Li, and K. W. Ross, "Understanding hybrid CDN-P2P: Why limelight needs its own red swoosh," in *Proc. 18th Int. Workshop Netw. Operating Syst. Support Digital Audio Video*, 2008, pp. 75–80.
[29] C. Gkantsidis and P. R. Rodriguez, "Network coding for large scale content distribution," in *Proc. IEEE 24th Annu. Joint Conf. IEEE Comput. Commun. Societies.*, 2005, pp. 2235–2245.
[30] D. Decasper and B. Plattner, "Dan: Distributed code caching for active networks," in *Proc. 17th Annu. Joint Conf. IEEE Comput. Commun. Societies*, 1998, pp. 609–616.
[31] J. West, "How open is open enough?: Melding proprietary and open source platform strategies," *Res. Policy*, vol. 32, no. 7, pp. 1259–1285, 2003.
[32] "Android lollipop," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Android_Lollipop
[33] "Android runtime (ART)," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Android_Runtime
[34] "ADBI - The android dynamic binary instrumentation toolkit," 2017. [Online]. Available: https://github.com/crmulliner/adbi
[35] L. Reyzin and N. Reyzin, "Better than biba: Short one-time signatures with fast signing and verifying," in *Proc. Australasian Conf. Inf. Security Privacy*, 2002, pp. 144–153.
[36] S. Seys and B. Preneel, "Power consumption evaluation of efficient digital signature schemes for low power devices," in *Proc. IEEE Int. Conf. Wireless Mobile Comput., Netw. Commun.*, 2005, pp. 79–86.
[37] S. Yu, C. Wang, K. Ren, and W. Lou, "Achieving secure, scalable, and fine-grained data access control in cloud computing," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.
[38] D. Zhang, Y. Qiao, L. She, R. Shen, J. Ren, and Y. Zhang, "Two time-scale resource management for green internet of things networks," *IEEE Internet Things J.*, vol. 6, no. 1, pp. 545–556, Feb. 2019.
[39] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proc. 21st USENIX Security Symp.*, 2012, pp. 569–584.
[40] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, 2014, Art. no. 5.
[41] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, 2014.
[42] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, 2004.
[43] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: Making smartphones last longer with code offload," in *Proc. 8th Int. Conf. Mobile Syst. Appl. Services*, 2010, pp. 49–62.
[44] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign Syst. Synthesis*, 2010, pp. 105–114.
[45] "Monsoon power monitor," 2017. [Online]. Available: https://www.msoon.com/LabEquipment/PowerMonitor/
[46] "2016 mobile marketing statistics compilation," 2016. [Online]. Available: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/
[47] G. Chen, B.-T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and R. Chandramouli, "Studying energy trade offs in offloading computation/compilation in java-enabled mobile devices," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 9, pp. 795–809, Sep. 2004.
[48] "The chromium projects," 2017. [Online]. Available: https://www.chromium.org/chromium-os
[49] "Google web store for chromium OS," 2017. [Online]. Available: https://chrome.google.com/webstore/category/apps
[50] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 29–42.
[51] T. Kelly, "Scalable TCP: Improving performance in highspeed wide area networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, pp. 83–91, 2003.
[52] "IEEE 802.11," 2017. [Online]. Available: https://en.wikipedia.org/wiki/IEEE_802.11
[53] M. Ali, "Green cloud on the horizon," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2009, pp. 451–459.
[54] A. Ahmed, A. Nasser, and S. Mohamed, "Mobile Cloud Computing: advantage, disadvantage and open challenge," in *Proc. 7th Euro American Conf. Telematics Inf. Syst.* 2014, p. 21.
[55] J. H. Christensen, "Using restful web-services and cloud computing to create next generation mobile applications," in *Proc. 24th ACM SIGPLAN Conf. Companion Object Oriented Program. Syst. Languages Appl.*, 2009, pp. 627–634.
[56] L. Liu, R. Moulic, and D. Shea, "Cloud service portal for mobile device management," in *Proc. IEEE 7th Int. Conf. e-Bus. Eng.*, 2010, pp. 474–478.
[57] R. K. Balan, D. Gergle, M. Satyanarayanan, and J. Herbsleb, "Simplifying cyber foraging for mobile devices," in *Proc. 5th Int. Conf. Mobile Syst. Appl. Services*, 2007, pp. 272–285.
[58] M. Shiraz, A. Gani, A. Shamim, S. Khan, and R. W. Ahmad, "Energy efficient computational offloading framework for mobile cloud computing," *J. Grid Comput.*, vol. 13, no. 1, pp. 1–18, 2015.
[59] K. Naik, "A survey of software based energy saving methodologies for handheld wireless communication devices," Dept. Electrical Comput. Eng., Univ. Waterloo, Waterloo, ON, Rep. 2010-13, 2010.
[60] S. Rosen, H. Yao, A. Nikravesh, Y. Jia, D. Choffnes, and Z. M. Mao, "Demo: Mapping global mobile performance trends with mobilyzer and mobiperf," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2014, pp. 353–353.
[61] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao, "Mobilyzer: An open platform for controllable mobile network measurements," in *Proc. 13th Annu. Int. Conf. Mobile Syst., Appl. Serv.*, 2015, pp. 389–404.
[62] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 27–38.
[63] J. Jeon, K. K. Micinski, and J. S. Foster, "Symdroid: Symbolic execution for dalvik bytecode," Department of Computer Science, University of Maryland, College Park, no. CS-TR-5022, Jul. 2012.
[64] "Smali: An assembler/disassembler for Android's dex format," 2016. [Online]. Available: https://github.com/JesusFreke/smali
[65] A. Desnos and G. Gueguen, "Android: From reversing to decompilation," in *Proc. Black Hat Abu Dhabi*, 2011, pp. 77–101.

Chao Wu received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2017, and the BEng degree in software engineering from Southeast University, Nanjing, China, in 2012. He is a postdoc in Tsinghua University. His research interests include mobile computing systems, edge computing and networked systems with a focus on operating system principle, performance optimization, architecture design. He is a member of the IEEE.

Lan Zhang received the bachelor's degree from the School of Software, Tsinghua University, China, in 2007, and the PhD degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2014. She is currently a research professor with the School of Computer Science and Technology, University of Science and Technology of China. Her research interests span privacy protection, secure multi-party computation and mobile computing, etc. She is a member of the IEEE

Zhenhua Li received the BSc and MSc degrees in computer science and technology from Nanjing University, in 2005 and 2008, and the PhD degree from Peking University, in 2013. He is an associate professor with the School of Software, Tsinghua University. His research areas mainly consist of cloud computing/storage, content distribution, and mobile Internet. He is a member of the IEEE.



Qiushi Li received the BEng degree from the University of Electronic Science and Technology of China, in 2017, and he is currently working toward the PhD degree at the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include mobile computing systems and applications.

Yaoxue Zhang received the BS degree from the Northwest Institute of Telecommunication Engineering, China, and the PhD degree in computer networking from Tohoku University, Japan in 1989. His major research interests include computer networking, operating systems, ubiquitous computing, big data driven service optimisation, and mobile networks. He serves as an associate editor of the *Chinese Journal of Computers* and the *Chinese Journal of Electronics*. He is a senior member of the IEEE.