A Microservice Architecture for Online Mobile App Optimization

Yixue Zhao
University of Southern California
yixue.zhao@usc.edu

Nenad Medvidovic
University of Southern California
neno@usc.edu

***Abstract*—A large number of techniques for analyzing and optimizing mobile apps have emerged in the past decade. However, those techniques' components are notoriously difficult to extract and reuse outside their original tools. This paper introduces MAOMAO, a microservice-based reference architecture for reusing and integrating such components. MAOMAO's twin goals are (1) adoption of available app optimization techniques in practice and (2) improved construction and evaluation of new techniques. The paper uses several existing app optimization techniques to illustrate both the motivation behind MAOMAO and its potential to fundamentally alter the landscape in this area.**

I. INTRODUCTION

Over the past decade, mobile computing devices have become dominant [1] and this trend is bound to continue into the foreseeable future. New technologies invariably bring challenges that require researchers' attention, such as the problems in the mobile domain that affect mobile app performance [2]–[8], energy use [9]–[15], and security [16]–[22].

We have studied the recent work in this domain to identify emerging research trends as well as problems that remain unaddressed. We have found that the research in the mobile app domain is still at a relatively early stage, and that there is a pronounced gap between research and practice. The majority of existing work still focuses on *identifying* problems, such as detecting performance bugs [3], [4], security vulnerabilities [16], [17], and energy hotspots [9], [15], while techniques for *solving* them are invariably left to future work.

As an illustration, 48 papers have dealt with mobile apps in the last five ICSEs (2014–2018). However, only 7 of those papers propose a technique that aims to optimize an app to address an identified problem. Even then, these techniques are usually evaluated on limited numbers of real apps, making their practical applicability unclear. For example, PALOMA [2] was evaluated on 32 apps and Bouquet [10] on only 5. Furthermore, these techniques are hard to adopt in practice because they usually involve non-trivial steps, such as advanced program analysis, that are likely to have a prohibitively steep learning curve for most app developers.

To identify the reasons behind the dearth of app optimization techniques and their lack of adoption in practice, we contacted the authors of several techniques and studied published causes of research-industry barriers [23]–[26]. We discovered several common themes. In the research community, (1) it is challenging to find subject apps that fit a given target problem; (2) app optimization techniques are usually built on top of research tools that have limited documentation and technical support; (3) it is hard to simulate real-world scenarios in a lab environment. From the practitioner's side, (1) it is hard to find research techniques that solve the exact problems a developer faces; (2) the research techniques are usually not evaluated in large-scale, real scenarios, rendering any claims unconvincing; (3) research techniques usually have limited documentation, making them difficult to adopt by app developers with little-to-no knowledge in a specific research area.

We believe the fundamental problem behind this gap is the lack of a standard protocol to guide the development of research techniques and to connect developers and researchers in a way that leverages each side's expertise. Specifically, individual app optimization techniques are designed in ad-hoc ways that hinder their reusability and composability. To address the problem, we propose a **M**icroservice **A**rchitecture for **O**nline **M**obile **A**pp **O**ptimization (MAOMAO) and a corresponding **M**icroservice **R**epository (MR). MAOMAO is a reference architecture for mobile app optimization techniques that is intended to be comprehensive in scope, but simple enough to be easily extensible. MR is a cloud-based repository to deploy MAOMAO-compatible techniques that connects researchers and developers by providing a shared baseline.

In this paper, Section II summarizes representative existing techniques and describes their (often missed) reuse opportunities. Section III introduces MAOMAO and discusses how existing techniques can be migrated to it. Section IV elaborates our vision for adopting MAOMAO in practice. Section V provides concluding thoughts and outlines the future work.

II. EXISTING TECHNIQUES

In this section, we introduce several independently developed techniques that focus on different problems in the mobile domain. We highlight each technique's major components to illustrate the potential (and, in practice, often missed) reuse opportunities. We will use these as well as other existing techniques to demonstrate how MAOMAO's architecture can integrate disparate existing solutions (Section III) and facilitate their reuse by both developers and researchers (Section IV).

PALOMA [2] reduces app latency by prefetching HTTP requests, via four major components: (1) *String Analyzer* identifies suitable HTTP requests for prefetching by interpreting their URL values; (2) *Callback Analyzer* detects the program points to issue prefetching requests; (3) *Instrumenter* uses the above information to produce a prefetching-enabled app; (4) at app runtime, the instrumented app triggers PALOMA's *Proxy* to issue prefetching requests and cache prefetched responses.

IMP [27] is a cost-benefit analysis that decides when and how much data to prefetch in an app, via three major components: (1) *API Support* provides "hints" on what to prefetch; (2) *Monitor* monitors mobile device's network bandwidth, data usage, and battery status; (3) *Prefetcher* adapts prefetching strategies based on the "hints" and runtime resource usage.

Bouquet [10] bundles HTTP requests to reduce the energy consumption of an app, via three major components: (1) *Detector* of Sequential HTTP Request Sessions (SHRSs), where triggering the first request implies the following requests will also be made; (2) *Bundling Analyzer* generates code to bundle each SHRS; (3) *Proxy* intercepts HTTP requests and runs the bundling code to return corresponding SHRS responses.

Many existing app optimization techniques focus on security. We highlight three representative examples. **IccTA**'s [17] *Taint Analyzer* detects privacy leaks among an app's components. **SEALANT**'s [16] *Analyzer* also identifies such leaks, while its *Interceptor* manages inter-app interactions to block the leaks. **ApkCombiner**'s [18] *Combiner* compiles multiple apps together to support inter-app privacy leak detection.

We see a notable reuse opportunity among these techniques. For instance, PALOMA and Bouquet can reuse IMP's *Monitor* and *Prefetcher* to dynamically adapt their strategies for issuing HTTP requests based on the runtime resource constraints. Bouquet's *Detector* can reuse PALOMA's *String Analyzer* to interpret the URL values when identifying SHRSs. PALOMA currently targets individual apps, but ApkCombiner's *Combiner* would enable PALOMA to prefetch HTTP requests across apps. In another scenario, SEALANT's *Analyzer* and IccTA's *Taint Analyzer* may be employed in tandem, either to directly compare their results (benefiting researchers) or to leverage their respective strengths (benefiting app developers).

However, reusing and combining existing research techniques is not a simple task in practice: their internal designs may not be properly modularized, their implementations may not be publicly available, and their documentation may be inadequate. Reuse and combination of different techniques' capabilities currently tends to require close communication with the authors of a given technique. This is time-consuming and unpredictable, resulting in regularly missed reuse opportunities and duplication of work. As a result, the above techniques have been successfully used in tandem in only two instances to our knowledge: SEALANT uses IccTA's *Taint Analyzer* to evaluate its own *Analyzer*'s accuracy, while ApkCombiner reuses IccTA to detect inter-app privacy leaks. In the latter case, ApkCombiner [18] and IccTA [17] share authors, which only further reinforces our point.

III. MAOMAO

We design MAOMAO based on the existing techniques, such as those highlighted above, and our own experience in the mobile domain. Our aim is to render reusable components at a proper granularity that can, both, serve as a roadmap for future techniques and improve the reusability of existing techniques. In this section, we introduce the design of MAOMAO's architecture (Section III-A) and elaborate how existing techniques can be integrated with MAOMAO (Section III-B).

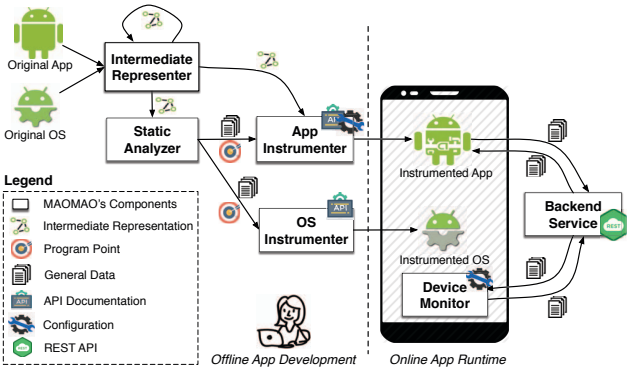

Fig. 1. MAOMAO's six reference components and overall workflow

A. MAOMAO's Design

MAOMAO's design is based on the widely-adopted microservice architectural style because (1) it helps to decouple potentially complex functionality into lightweight, "black-box" microservices, which are easy to understand and adopt by developers in practice [28]; (2) existing mobile techniques tend to comprise clearly separable and often reusable components, and the microservice style would make it easier to reuse such components across techniques; (3) the microservice style allows components (i.e. microservices) to be implemented in different programming languages with different technologies, which suits the heterogeneity of the mobile domain.

As Fig. 1 shows, MAOMAO's reference architecture consists of six components, i.e., microservices. An individual app optimization technique can consist of one or more of the reference components. For example, IccTA [17] only has the *Intermediate Representer* and *Static Analyzer*.

Intermediate Representer takes an app or the Operating System (OS), e.g., Android framework, as the input and produces an Intermediate Representation (IR) for *Static Analyzer* to analyze. IR can be used by other Intermediate Representer to build new IR. For example, tool-specific IR is usually built on top of fundamental IRs, such as Abstract Syntax Tree (AST), Control Flow Graph (CFG) of an app. GATOR [29] has an *Intermediate Representer* to produce Callback Control Flow Graph (CCFG), which is a tool-specific IR that uses CFG.

Static Analyzer analyzes the IR to extract useful information that can be used in other components, such as the program point to be instrumented that will be used to instrument the app or the OS. For instance, PerfChecker [3] has a *Static Analyzer* to detect performance bugs, which can be used by app developers directly or reused by follow-up techniques to fix the bugs based on the bug locations (i.e., program point).

App Instrumenter instruments the app code and transforms the original app, usually based on the information extracted from the *Static Analyzer*. The *App Instrumenter* can be categorized into *Automatic App Instrumenter* (AAI) or *Manual App Instrumenter* (MAI), and it usually needs to be configured so that the instrumented app can interact with other specific components at runtime, such as *Backend Service*. An AAI instruments the app without developer's involvement, e.g., PALOMA's [2] *Instrumenter* is an AAI used to

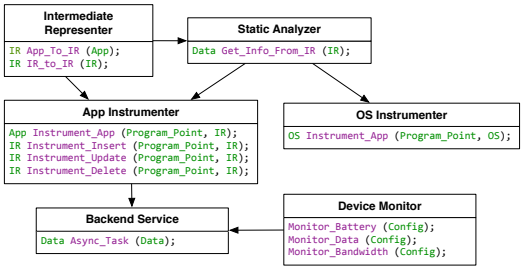

Fig. 2. MAOMAO's reference components and their reference APIs

enable prefetching based on the information extracted from PALOMA's *Static Analyzer*s. On the other hand, a MAI provides APIs for developers to manually modify their code.

OS Instrumenter is similar to *App Instrumenter*, but it instruments the OS (e.g., Android) instead of the app. *OS Instrumenter*s can also be categorized into *Automatic OS Instrumenter*s (AOSI) and *Manual OS Instrumenter*s (MOSI). For instance, SEALANT's *Interceptor* is a MOSI that extends the Android framework to block malicious intents at runtime.

Device Monitor observes the device-level conditions at app runtime. It is typically used to balance the quality-of-service (QoS) trade-offs since mobile devices are resource-constrained. Similar to the *App Instrumenter*, it also needs to be configured in order to interact with other components at runtime, such as the *Backend Service*. For instance, IMP's *Monitor* is a *Device Monitor* targeting battery life, data usage, and network bandwidth, that interacts with its *Prefetcher*.

Backend Service contains the ancillary functionalities that are triggered at app runtime. It interacts with the instrumented app and the *Device Monitor* via lightweight protocol, e.g., REST APIs [30]. The ancillary functionalities are usually triggered by specific information sent from the instrumented app or the *Device Monitor*. For instance, IMP's *Prefetcher* is a *Backend Service* that adapts its prefetching strategies according to the device's QoS conditions sent by its *Monitor*.

Fig. 2 shows the reference APIs for each reference component that aims to aid the design and implementation of MAOMAO, and a concrete example will be shown in Section III-B.

B. Migration of Existing Techniques to MAOMAO

We hypothesize that designing new techniques using MAOMAO's microservice architecture will be more straightforward than migrating an existing technique. For this reason, in this section we illustrate how the latter process can be approached using examples from Sections II and III-A. Specifically, we studied the designs of the existing techniques as well as their available open-source implementations to establish that they can be ported to MAOMAO's architecture.

Table I shows the mapping between the components in the existing techniques and MAOMAO's components. We use PALOMA [2] as an example to explain the details of the mapping. Fig. 3 shows the class diagram of PALOMA when migrated to MAOMAO following the reference components and APIs shown in Fig. 2. PALOMA was selected because it was recently published and it contains more components to be migrated to MAOMAO's architecture than the other techniques, which have as few as a single relevant component.

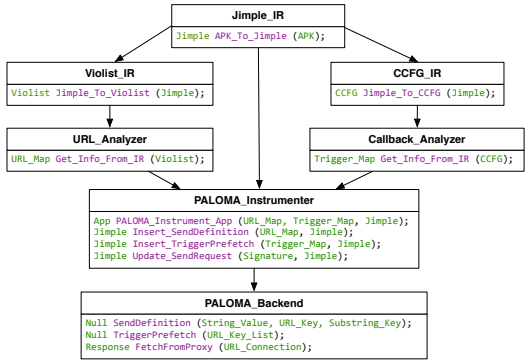

Fig. 3. The class diagram of PALOMA in MAOMAO

PALOMA's *String Analyzer* leverages an external string analysis tool, Violist [31], to identify the string values of URLs in an app, and outputs a `URL Map`. Violist has a proprietary intermediate representation (IR) of the control- and data-flow relationships among the string variables and string operations. Violist's IR is transformed from Jimple [32], which is a fundamental IR for representing Java/Android programs. Violist analyzes the string values at given program points based on its IR. With MAOMAO, PALOMA's *String Analyzer* will be implemented as two *Intermediate Representer* microservices (`Jimple_IR`, `Violist_IR`) and one *Static Analyzer* microservice (`URL_Analyzer`) as shown in Fig. 3. `Jimple_IR`'s `APK_To_Jimple` API is an implementation of *Intermediate Representer*'s `App_To_IR` (Fig. 2), where `Jimple` is an instance of `IR`, and `APK` [33] is an instance of `App`. Similarly, `Violist_IR`'s `Jimple_To_Violist` is an implementation of *Intermediate Representer*'s `IR_To_IR`, where `Violist` and `Jimple` are both instances of `IR`. `URL_Analyzer`'s `Get_URLMap_From_Violist` is an implementation of *Static Analyzer*'s `Get_Info_From_IR`.

PALOMA's *Callback Analyzer* leverages an external callback analysis tool, GATOR [29], to identify the data prefetching points in an app, and generates a `Trigger Map`. Specifically, the *Callback Analyzer* relies on the CCFG defined by GATOR. With MAOMAO, the *Callback Analyzer* is decomposed into two microservices: an *Intermediate Representer* (`CCFG_IR`) that outputs the CCFG by reusing `Jimple_IR`, and a *Static Analyzer* (`Callback_Analyzer`) that outputs the `Trigger_Map` for instrumenting prefetching functions at given program points based on the CCFG.

PALOMA's *Instrumenter* takes as inputs the `URL_Map`, the `Trigger_Map`, and the `Jimple`, and transforms

TABLE I
MAPPINGS BETWEEN EXISTING COMPONENTS IN EXISTING TECHNIQUES (SECTION II) AND MAOMAO'S COMPONENTS (SECTION III-A)

Existing Technique	Existing Component	MAOMAO's Component
PALOMA [2]	*String Analyzer*	Intermediate Representer + Static Analyzer
	Callback Analyzer	Intermediate Representer + Static Analyzer
	Instrumenter	Intermediate Representer + (Automatic) App Instrumenter
	Proxy	Backend Service
IMP [27]	*API Support*	(Manual) App Instrumenter
	Monitor	Device Monitor
	Prefetcher	Backend Service
Bouquet [10]	*Detector*	Intermediate Representer + Static Analyzer
	Bundling Analyzer	Static Analyzer + (Automatic) App Instrumenter
	Proxy	Backend Service
IccTA [17]	*Taint Analyzer*	Intermediate Representer + Static Analyzer
SEALANT [16]	*Analyzer*	Intermediate Representer + Static Analyzer
	Interceptor	(Manual) OS Instrumenter
ApkCombiner [18]	*Combiner*	(Automatic) App Instrumenter

the original app to a prefetching-enabled app with three instrumentation functions. With MAOMAO, PALOMA's *Instrumenter* will be one *App Instrumenter* (`PALOMA_Instrumenter`) that instruments the app based on the outputs from `URL_Analyzer`, `Callback_Analyzer`, and `Jimple_IR` as shown in Fig. 3.

Finally, PALOMA's *Proxy* interacts with the instrumented app at runtime via the three instrumented functions: (1) it updates the `URL Map` with the data sent by `SendDefinition`; (2) it prefetches HTTP requests triggered by `TriggerPrefetch`; and (3) it redirects the HTTP requests to get the response from a cache triggered by `FetchFromProxy`. Migrating PALOMA's *Proxy* to MAOMAO is straightforward: it will be designed as a single *Backend Service* microservice (`PALOMA_Backend`), with three REST APIs to represent the three instrumented functions.

Other existing mobile computing techniques would be redesigned (and subsequently reimplemented) in an analogous fashion. As mentioned previously, new techniques would follow MAOMAO's reference architecture and rely on its APIs from the get-go.

IV. MAOMAO's envisioned adoption

MAOMAO's architecture allows app optimization techniques to be decomposed into lightweight reusable microservices with a standard workflow, enabling their use by both developers and researchers. Specifically, MAOMAO alleviates the problem of reusing often incompatible capabilities from disparate research techniques.

To realize MAOMAO's potential in practice, we propose a *Microservice Repository* (MR), that aims to connect developers and researchers together by providing and enforcing a shared baseline. MR is a cloud-based repository that consists of a *Service Request Pool* (SRP) and a *Microservice Pool* (MP). SRP stores developers' requests for their desired services. MP stores and provides access to the available microservices and their corresponding API documentation.

We use mobile app security techniques—IccTA [17], SEALANT [16], and APKCombiner [18]—to demonstrate how developers and researchers can benefit from MAOMAO and MR. We choose security because it has attracted the greatest attention among researchers in the mobile domain.

As Table I shows, IccTA's *Taint Analyzer* is decompsosed into *Intermediate Representer* (IR) and *Static Analyzer* (SA) microservices in MAOMAO's architecture. Similarly, SEALANT consists of IR, SA, and *Manual OS Instrumenter* (MOSI) microservices. Finally, ApkCombiner becomes an *Automated App Instrumenter* (AAI) microservice.

The six MAOMAO microservices in the three techniques will be deployed to MR's MP, along with their corresponding API documentation. We discuss several representative use cases of developers and researchers using these services.

1) Both developers and researchers can search the MR to find their desired microservices in a specific domain (e.g., mobile security domain).
2) IccTA's authors can find ApkCombiner's AAI in the MR and extend IccTA's SA to detect *inter-app* privacy leaks by following the API documentation of ApkCombiner's AAI. Then, IccTA's optimized inter-app SA can be deployed as a new microservice to the MR.
3) SEALANT's authors can extend its SA in the same manner as IccTA. SEALANT and IccTA can then use each other's IR and SA microservices to compare the two solutions. Moreover, since SEALANT's MOSI outputs an instrumented OS to block privacy leaks, it can be used by IccTA to "upgrade" from detection to optimization.
4) A phone manufacturer's engineers can find SEALANT's MOSI in the MP and follow its APIs to customize the OS to block privacy leak on their phones.
5) If developers cannot find a desired microservice in the MR, they can submit a service request to the SRP to describe their needs (e.g., a request for a performance bottleneck detection service). They can optionally attach a benchmark app that has the relevant issue.
6) Researchers can search the SRP to find reported needs in their domain of interest and possibly obtain the corresponding testing data (e.g., using submitted benchmark apps to evaluate a performance bug detection technique).

MAOMAO's microservices and MR allow researchers to track the real-world needs and developers to adopt research techniques readily by invoking lightweight APIs. An added advantage is that the microservices are deployed on the cloud and do not introduce significant overhead on the client apps deployed on resource-constrained mobile devices. Researchers can also dynamically update their microservices without requiring modifications to the app code. In addition, the testing data provided by developers in the SRP can serve as benchmarks for comparing different techniques in the same domain. Once the microservices are adopted by developers, the underlying research techniques will be evaluated in the real world with real users, providing insights and incentives for researchers to improve their techniques.

V. Conclusion and Future Work

We introduce MAOMAO, a reference architecture for mobile app optimization techniques to guide the design of future techniques in order to improve their reusability and extensibility, with a corresponding Microservice Repository (MR) to deploy MAOMAO-compatible techniques. Together, the two improve the availability and practicality of research techniques, and bridge the gap between researchers and developers. Our preliminary work provides evidence of MAOMAO's and MR's viability, and also shows several future research directions in order to adopt MAOMAO in practice, such as ensuring privacy of any data (e.g., app usage) submitted to MR, scalability to large numbers of researchers and developers, and standardizing API documentations. As early versions of MAOMAO and MR are deployed and adopted, this scope will grow to include capabilities such as recommenders of related microservices based on certain service requests and an access control model to enable fine-grained data sharing.

REFERENCES

[1] S. KEMP, "Digital in 2018," January 2018. [Online]. Available: https://wearesocial.com/blog/2018/01/global-digital-report-2018

[2] Y. Zhao *et al.*, "Leveraging program analysis to reduce user-perceived latency in mobile applications," in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2018.

[3] Y. Liu *et al.*, "Characterizing and detecting performance bugs for smartphone applications," in *International Conference on Software Engineering (ICSE)*, 2014.

[4] Y. Liu, "Automated analysis of energy efficiency and execution performance for mobile applications," Ph.D. dissertation, Hong Kong University of Science and Technology, 2015.

[5] Y. Wang, X. Liu, D. Chu, and Y. Liu, "Earlybird: Mobile prefetching of social network feeds via content preference mining and usage pattern analysis," in *Proceedings of the 16th ACM International Symposium on Mobile Ad Hoc Networking and Computing*. ACM, 2015, pp. 67–76.

[6] Y. Zhang, C. Tan, and L. Qun, "Cachekeeper: a system-wide web caching service for smartphones," in *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*. ACM, 2013, pp. 265–274.

[7] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 113–126.

[8] A. Pathak *et al.*, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012.

[9] A. Banerjee *et al.*, "Detecting energy bugs and hotspots in mobile apps," in *Symposium on Foundations of Software Engineering (FSE)*, 2014.

[10] D. Li *et al.*, "Automated energy optimization of http requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, May 2016.

[11] A. Banerjee and A. Roychoudhury, "Automated re-factoring of android apps to enhance energy-efficiency," in *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*. IEEE, 2016.

[12] Y. Liu *et al.*, "Greendroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, no. 1, 2014.

[13] Y. Lyu, D. Li, and W. G. Halfond, "Remove rats from your code: automated optimization of resource inefficient database writes for mobile applications," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 310–321.

[14] Y. Liu, C. Xu, and S.-C. Cheung, "Diagnosing energy efficiency and performance for mobile internetware applications: Challenges and opportunities," *IEEE Software*, 2015.

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

[16] Y. K. Lee *et al.*, "A sealant for inter-app security holes in android," in *39th International Conference on Software Engineering (ICSE)*, 2017.

[17] L. Li *et al.*, "Iccta: Detecting inter-component privacy leaks in android apps," in *37th International Conference on Software Engineering*, 2015.

[18] L. Li, A. Bartel *et al.*, "Apkcombiner: Combining multiple android apps to support inter-app analysis," in *IFIP Int'l Information Security Conference*, 2015.

[19] V. Avdiienko *et al.*, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015.

[20] M. Hammad *et al.*, "Self-protection of android systems from inter-component communication attacks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018.

[21] 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.

[22] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[23] M. Harman and P. OHearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis."

[24] K. Sherif and A. Vinze, "Barriers to adoption of software reuse: A qualitative study," *Information & Management*, vol. 41, no. 2, 2003.

[25] J. R. Cordy, "Comprehending reality-practical barriers to industrial adoption of software maintenance automation," in *Program Comprehension, 2003. 11th IEEE International Workshop on*. IEEE, 2003.

[26] S. T. Redwine Jr and W. E. Riddle, "Software technology maturation," in *Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 1985.

[27] B. D. Higgins *et al.*, "Informed mobile prefetching," in *10th international conference on Mobile systems, applications, and services*, 2012.

[28] M. Villamizar *et al.*, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 2015.

[29] S. Yang *et al.*, "Static control-flow analysis of user-driven callbacks in android applications," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015.

[30] R. T. Fielding and R. N. Taylor, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, USA, 2000, vol. 7.

[31] D. Li *et al.*, "String analysis for java and android applications," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[32] A. Bartel *et al.*, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN Int'l Workshop on State of the Art in Java Program analysis*, 2012.

[33] "What is an apk file?" 2018. [Online]. Available: https://fileinfo.com/extension/apk