Mobile Applications Need Targeted Micro-Updates

Alvin Cheung* Lenin Ravindranath* Eugene Wu*
Samuel Madden Hari Balakrishnan
MIT CSAIL

1 INTRODUCTION

Smart-phone applications ("apps") run across a wide range of environmental conditions, locations, and hardware platforms. They are often subject to an array of interactions that are hard or impossible for developers to emulate or even anticipate during testing. Once an application is released, feedback obtained from users and from analytics over usage and performance data result in further modifications. Many of these changes are relatively small, and can often be parameterized.

We call such post-deployment updates *micro-updates*. Given the diversity of environments in the mobile ecosystem, there is often no single optimal parameter value for all running instances of an application. Application behavior and micro-updates need to be *targeted* to different sets of users. Micro-updates can range from simple updates to static configurations, program constants, and UI properties to changing the behavior of a few isolated routines. Examples include changing a hard-coded timeout parameter to accommodate slow network connectivity, reducing the location sampling interval on certain devices to control battery drain, adapting certain UI properties to accommodate new devices with different screen resolutions, adding a resource module to support new locales, and changing the behavior of a few routines to avoid an app crash while running on a specific version of the OS.

Today, targeted micro-updates can be done in two ways. The first is to modify the app source code, rebuild it, and put out an update in the app store. Although some app stores such as Google Play enable automatic updates, none of the current app stores provide a way to *target* updates to a particular set of users. Hence, developers need to pack every possible program behavior into a single app that deals with all sorts of conditions that can arise in the wild. The more differentiated conditions and desired behavior becomes, the more difficult it is to manage updates and application logic. Moreover, app stores such as the iOS AppStore impose a manual verification and update installation process for updating the app, which can significantly delay the update to users. And even when it is released, only a fraction of users might actually install the update, creating significant versioning problems.

The second option to perform a micro-update is for the developer to write the application so that specific parts are periodically re-loaded from a remote server. When the app needs to be updated, the developer modifies the application logic on the server. Targeted updates can be provided by monitoring and collecting appropriate data from the users and indicating which devices should install an update in the update itself.

Unfortunately, building and maintaining such a custom infrastructure for monitoring and updating is very difficult because the developer must now 1) maintain a server to host the updates, 2) mix the update and application logic in the code base, 3) manage performance problems that can arise when devices contact the remote server for updates, 4) develop mechanisms to propagate selective updates, and 5) manage which devices should receive which updates.

In this paper, we propose Satsuma, a service infrastructure and framework for pushing updates that significantly reduce the barrier for developers to micro-update their application in a faster and more targeted fashion. Using Satsuma requires low development effort and no modification to the program logic. Developers simply annotate parts of the application where they anticipate updates after deployment. After deployment, developers can use Satsuma to monitor and update portions of the application and apply updates to targeted sets of users. Satsuma also includes a dynamic monitoring framework to target updates across sets of users.

Note that Satsuma does not decrease the security of mobile application ecosystems. Using Satsuma to launch malicious attacks, for instance by creating an alarm clock application and subsequently releasing a malicious micro-update, is already possible in the current ecosystem via standard app update mechanisms. Instead, Satsuma's goal is to make application and configuration management easier post-deployment.

In summary, this paper makes the following contributions:

- We list a number of use cases that are enabled by providing micro-updating capability for mobile apps.
- We describe a declarative device targeting service for developers to push micro-updates to devices after deployment.

*authors contributed equally

- We describe how code annotations can be delegated, so that other parties can also push micro-updates in a limited fashion.

In the following we first describe motivating use cases for micro-updates, followed by a description of Satsuma. We conclude by outlining future research directions.

2 USE CASES

In this section, we describe several use cases that the Satsuma micro-updating framework enables.

2.1 Targeted Fine Tuning

After an application is released, developers use Satsuma to fine-tune their application behavior based on how the application runs in the hands of real users — this fine-tuning can vary between users.

For example, suppose a developer of a location tracking application is not able to determine the optimal GPS sampling rate with an acceptable energy consumption across different devices. He uses a default value that is optimized in the lab setting but uses Satsuma to mark the sampling rate parameter as micro-updatable. Once the application is released, he soon learns that the battery drains unacceptably quickly on a particular phone model, and he wants to reduce the sampling rate to save energy without compromising the data resolution obtained from other devices that do not have the issue. The developer uses Satsuma to perform a targeted micro-update to reduce the sampling rate only on the anomalous devices.

In addition to updating parameters and configurations, developers can also update parts of the program logic after deployment, for instance tailoring the app for users in different geographic regions.

2.2 Experimental Testing

User experimentation, such as A/B testing, is commonly used in web development and marketing to test variations of the application to improve user experience and other metrics. For example, a developer may want to know which of two different UIs is more effective for users. Rather than picking one for the entire deployment, she can deploy the application and use Satsuma to run A/B tests. For instance, she can update 10% of the users with a new UI and gather feedback. Based on the feedback, the developer can decide to update more or all its users with the new UI. Similar A/B tests can be done when choosing between performance configurations, application features, interface designs, and user-specific customizations.

2.3 Dynamic Monitoring

In addition to updating application behavior, Satsuma can be used to passively monitor application behavior in the wild. Today, developers are limited to analytics and performance monitoring systems [2, 16] that statically instrument their application for passive monitoring. Such instrumentation is typically unmodifiable after deployment. When developers want to collect more data, they must release a new application binary through the app store.

Satsuma lets developers dynamically instrument the application by simply marking a monitoring function as updatable, and injecting new instrumentation code to execute on apps that are currently running on the devices.

2.4 Bug-Fixing

It is hard to anticipate where bugs will arise in code. If a bug arises in a small part of the code, the developer must update the entire application and hope the users will install the update.

Alternatively, the developer can use Satsuma and mark all major code fragments as updatable. When a bug arises, he can remotely update and correct that portion of the app. As we discuss in Sec. 4, as more code fragments are annotated with Satsuma, there can be a runtime overhead. However, we believe that there are still certain scenarios where this usage model is beneficial. For instance, a developer marks everything only during beta-testing with a small to moderate set of users. As compared to manually updating and distributing the application for each bug fix during beta-testing, Satsuma can dramatically reduce the management difficulty and the latency of the end-to-end debugging cycle.

3 DESIGN

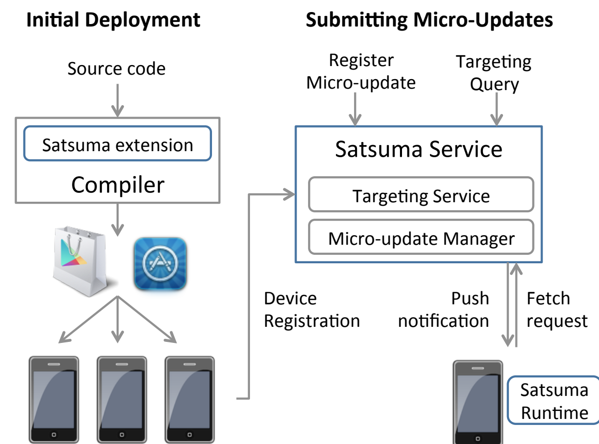

Figure 1: Satsuma architecture

Fig. 1 shows the Satsuma architecture and the two primary control flows. Satsuma consists of three components: a **compiler extension** that takes in code with annotations denoting which parts of the app can be micro-updated, a **service** that manages micro-updates and executes targeted updates and queries, and a **runtime module** that is linked with the application.

The compiler initially processes the annotated source code to link the Satsuma Runtime and adds the necessary event handlers and instrumentation hooks. It then generates a deployable binary. The developer publishes the binary to the app store where users can download the application.

After the app is deployed, the developer changes previously annotated code fragments and re-compiles the code. The extension then creates a new micro-update that can be registered with the Micro-Update Manager. The Manager is a lower-level service that pushes notifications and handles micro-update download requests from the devices.

The Targeting Service executes high-level SQL-like queries from developers that collect data from the devices or push micro-updates to a targeted set of devices. Under the covers, the service pushes the appropriate micro-updates to devices in order to gather the queried data.

When users first run the application, the Satsuma Runtime registers itself with the Satsuma Service. It also manages micro-update downloads and provides convenience methods.

4 Satsuma Annotations

As mentioned in Sec. 3, the developer annotates her source code during development to describe where micro-updates can be applied. We briefly describe annotations for Java-based applications, then illustrate how micro-updates are pushed and executed on Android and other platforms. This section describes the low level mechanics to support micro-updates, and the next section describes the higher level targeting framework layered on top of these mechanisms.

Although some app stores enable automatic updates, they are not universally enabled because users must turn it on manually. Because of that, Satsuma does not rely on users obtaining micro-updates via such mechanism.

4.1 Using Annotations

Developers use the `@allowUpdate` annotation to allow a piece of code be micro-updatable after deployment. `@allowUpdate` annotations can be placed before static initialization blocks, function definitions, and class definitions. Each annotation indicates that the corresponding program fragments can be updated. For example, a developer who wants to fine-tune a `samplingRate` parameter and profile her activity detection algorithm post-deployment annotates her program as follows:

```
@allowUpdate static int samplingRate = 10;

@allowUpdate void detectUserActivity_preHook () {}
@allowUpdate void detectUserActivity_postHook () {}

public void detectUserActivity () {
  detectUserActivity_preHook();
  ...
  detectUserActivity_postHook();
  return; }
```

Given the above annotations, the developer can update the value of `samplingRate`, and can insert logging statements (e.g., for start and end times) into the intercept methods that are called before and after `detectUserActivity`.

While it may seem tempting to annotate every code block to maximize post-deployment update flexibility (for instance, in the case of pushing bug fixes), there are performance and manageability costs in doing so. As Sec. 4.3 illustrates, each annotation incurs a runtime overhead on non-Android platforms. In addition, while the Satsuma compiler includes utility flags (e.g., `-annotateAll`) to ease some of the developer's management burdens, there are many interface challenges as the code base, and the number of annotations, grows (to be discussed in Sec. 7).

4.2 Pushing Notifications

The developer pushes a micro-update by sending the update and a list of target device IDs to the Satsuma micro-update manager. The service compresses the update and pushes a notification to the listed devices. We re-use each mobile platform's push notification services, and encode the service IP and micro-update ID in the notification's payload.

4.3 Satsuma Mobile Runtime

The Android platform supports dynamic class reloading, thus updates can be applied by reloading the modified classes. Since the micro-updates are downloaded as data files, they must be re-loaded every time the application starts, however multiple micro-updates may be merged into a single load operation. The disadvantage of this approach is that the program will contain both the old and new instances of the modified classes. One means to circumvent this is to monitor all heap accesses, and ask the developer to provide conversion functions to convert objects between the old and new object representations [12].

Satsuma uses function stubbing on other platforms (e.g., iOS, Windows) that do not support class reloading. The idea is that during initial compilation, the compiler first rewrites all annotated methods with stubs. The stubs are implemented as remote procedure calls (RPC) hosted on the Satsuma server that forward the necessary state as arguments to the RPC call each time the annotated method is invoked. Micro-updates simply replace the implementation of the RPC on the server. This mechanism is also applicable to modifying definitions of literals. Unfortunately, RPCs can incur substantial overhead. Alternatively, an interpreter can be embedded within the Satsuma runtime, and micro-updates are then compiled to the interpreter's language. Compared to stubbing, the interpreter approach might have less overhead than invoking RPCs, and has been deployed in iOS applications [1].

5 Collecting Data for Targeting

This section describes the mechanisms to declaratively collect and query data gathered from the deployed applications, and perform targeted micro-updates using the same querying mechanisms after the updates have been submitted using the Satsuma service.

5.1 Language for Data Collection

Satsuma logically exposes the data that can be collected from the mobile devices as tables queryable using a SQL-like lan-

guage. The tables are "backed" by micro-updates that collect data in the table's schema. For convenience, Satsuma provides a set of pre-defined tables for device hardware, OS, ID and location information. For example, the following defines a `location` table of device location information using the familiar `create table` SQL syntax, and backs the table with the `Loc-update` micro-update:

```
create table location (UID deviceId,
  String city, String state, Time tstamp);
back table location with Loc-update;
```

`Loc-update` is written to send `location` table rows back to the server, and Satsuma ensures that the received rows have the same schema as the target table. The developer can now declaratively query this table. For instance, the following query collects and stores the city and state information from all deployed applications running on devices in California:

```
select city, state
from location
where state = "California"
```

If `location` is not being populated by an existing query, Satsuma will automatically push `Loc-update` to collect the data.

5.2 Language for Device Targeting

The developer expresses her request to push targeted micro-updates using a similar language as that for data collection. For example, the following query pushes the `NorCal-update` and `SoCal-update` to devices in San Francisco and Los Angeles, respectively:

```
push "NorCal-update" to (
  select deviceId from location
  where location.city = "San Francisco");

push "SoCal-update" to (
 select deviceId from location
 where location.city = "Los Angeles");
```

To execute this `push` query, Satsuma may need to first send `Loc-update` to collect location data, then filter the collected data so that `NorCal-update` and `SoCal-update` can be pushed to the correct devices. Note that the developer can push to all installed applications (e.g., in the case of bug fixes) using a `select *` query on the pre-defined device ID table.

5.3 Implementation

All data collection queries are either answered using pre-collected data, or are compiled into micro-updates that are pushed to the devices. Both data collection and `push` queries are subject to the same set of permissions that are initially granted by the user during application installation. Thus, in the example above, the application needs permission to read GPS sensor readings in order to collect location data, and the Satsuma compiler will warn the developer if that is not the case.

6 Delegating Micro-updates

In a mobile ecosystem where multiple parties develop different parts of an application, one party will want to *delegate* limited micro-update privileges to others. In this section, we present two common use cases, third-party libraries and multi-team development, that showcase this need. We then describe how to extend Satsuma annotations to support this requirement.

6.1 Use Cases

Applications commonly include third-party libraries such as advertising or utility SDKs. Consider Dave, who develops a remote logging library and wants to optimize the transmission frequency parameter to account for the device's battery life. However, each device consumes energy at different rates, so David adds an annotation so he can update the parameter post-deployment. Alice uses the library in her application and is happy that David will optimize the performance for her, but wants to ensure that David modify nothing else.

As as second example, large applications are composed of distinct components managed and micro-updated by separate teams. For example, one team may optimize the location estimation library, whereas the marketing or UX department may update the app's dashboard interface. It is important to reduce the chance that a micro-update will break the application by statically restricting the scope of possible updates.

6.2 Annotation Policies

The uses cases above suggest the need for access control and micro-update scope policies. We support this by extending `@allowUpdate` to accept a list of (user, scope) pairs that each describe the extent that the user can modify the annotated fragment. Only users explicitly listed can micro-update the fragment. We propose two possible scope values in the initial design:

- `all`: means that the user can perform any modification to the target program fragment, such as calling methods, deleting statements, or modifying class definitions.
- `sandboxed`: restricts the user to only be able to add statements without side-effects (and invoke other side-effects free methods), or that are specified in a white-list of methods with side-effects (e.g., gathering data from sensors).

For convenience, Satsuma provides a collection of white-lists that contain common methods for micro-update tasks such as sensor reading (e.g., GPS, accelerometer), logging, and data collection. The `sandbox` scope is expressive enough for the use cases above, yet effectively restricts micro-update abilities. In contrast, `all` is useful during beta testing.

6.3 Implementation

During initial compilation, the Satsuma compiler records all annotations and their corresponding program fragments. In

addition, it generates a key for each user encountered. The key can be subsequently distributed to the users.

To make a micro-update, the user submits her modified code and key to the compiler. If the modification passes all annotation constraints, the compiler creates modified class files to be submitted as micro-updates.

7 Research Opportunities

Beyond the descriptions above, we outline a number of future research directions.

Data collection privacy. Users of applications that allow micro-updates might be concerned about the type of data that can be collected. For instance, while the user might feel comfortable with a mapping application reading the GPS sensor to provide navigation directions, she might be concerned about sending all GPS readings to the developer, or the data collection code draining up her battery. One research direction would be to allow users specify privacy policies for data collection, and enforce them in the runtime.

Device targeting optimization and scalability. Having a declarative interface for device targeting raises interesting optimization opportunities. For instance, for the location data collection example discussed in Sec. 5.1, the runtime can collect the requested data by reading the GPS sensor on the device, or use other sensors to approximate location. The runtime can also vary the sensor sampling frequency based on the user's location. In addition, the runtime can combine micro-update and data collection requests. For instance, the data filtering that is part of the `push` query in Sec. 5.1 can be pushed into `Loc-update`, which effectively combines two micro-updates into one. It will be an interesting problem to build an optimizer for instrumentation instructions. Another potential research problem is to build a scalable storage system for the collection location data, as that would affect the the amount of time needed to push updates to all the intended devices.

Managing micro-update histories. Once developers target micro-updates at different subsets of the installation base, managing the different, possibly overlapping, versions of the code base quickly becomes overwhelming. In the case of using Satsuma for A/B testing, the developer may want to pick and combine the optimal results from multiple tests in the next major update of the application. However, if the targeted populations overlapped, then how can the collection of micro-updates be merged into a single coherent source tree for the next major release?

One model is to view this as a version control problem, where every micro-update is a new commit, and concurrent updates (to different target devices) are separate branches. The system can provide means for developers to merge and reconcile different updates into a new release, along with tracking the updates that have been received by each device. Developing a user interface that can easily facilitate such merges and device tracking would be interesting research.

Managing Annotations. Sec. 4 introduced a mechanism for developers to limit who and how micro-updates can be made. As the code base and number of third-party libraries used increases, however, it will be difficult for developers to manage the granted permissions. A research direction would be to build a management tool for visualizing and annotating source code repositories, and provide static guarantees about behavior of micro-updates.

Cross-platform micro-updates. In the modern mobile application ecosystem, the same application is typically deployed on multiple mobile platforms, with different code bases for each platform despite they all implement similar functionality. While there are programming systems that try to implement "write-once, deploy everywhere" for mobile platforms [6, 5], it will be interesting to investigate similar issues for applying micro-updates. Challenges include handling differences among platforms, and building an interface for developers to provide correspondences among the different application code bases.

8 Related Work

To our knowledge, Satsuma is the first proposal to build a system that pushes application updates onto deployed mobile applications, and lets developers control the users that can submit updates along with the expressiveness of the updates.

Dynamic code injection and instrumentation have been used to let developers detect events on mobile devices [17, 3], detect bugs [14], and declaratively profile non-mobile applications [9]. Satsuma proposes to leverage similar techniques to optimize instrumentation requests for mobile applications.

Satsuma's stubbing techniques are similar to code partitioning between mobile devices and cloud services [13, 10, 11] for the purposes of reducing resource or battery consumption. Our micro-update framework can be used to implement code offloading as well.

Annotations have been used to provide security in web applications [4]. Satsuma uses the same mechanism to limit the scope of and users that can micro-update an annotated declaration.

Finally, both commercial [8] and academia [15, 7, 12] are actively investigating techniques to dynamically update software on non-mobile platforms. The mobile platform, however, poses new challenges as compared to providing updates to desktop applications. For instance, mobile devices come in a much larger variety of hardware and software specifications than desktop machines. Because of that, the micro-update system will need to ensure the compatibility of any micro-updates that are pushed to the device, and warn the developer if it detects incompatibility on devices that have the application installed. In addition, as outlined by our use cases, developers will likely want to target devices to be updated based on device-specific attributes such as geographic location in addition to hardware profiles. Tracking locations for mobile devices is a challenging problem [18],

and such scenarios do not usually arise in desktop application updates.

9 Conclusion

In this paper, we introduced Satsuma, a system for pushing targeted micro-updates for mobile apps. We described several use cases that are unaddressed in the current mobile app development ecosystem, outlined the design of Satsuma and its declarative interface for targeting devices, and proposed a number of future research directions. We believe the ability to selectively push micro-updates to different subsets of devices has the potential to dramatically simplify mobile application development by enabling a variety of tasks, such as A/B testing, fine-grained device reconfiguration on targeted phone models or users, and rapid deployment of bug fixes.

10 Acknowledgements

The authors are grateful for the constructive feedback from the reviewers. This research is supported by Intel Corporation.

References

[1] Codea. `http://twolivesleft.com/Codea`.

[2] flurry. `http://www.flurry.com`.

[3] Microsoft on{x}. `http://www.onx.ms`.

[4] Oracle Java EE 5 Tutorial. `http://docs.oracle.com/javaee/5/tutorial/doc/bnbyl.html`.

[5] Phonegap. `http://www.phonegap.com`.

[6] TouchDevelop. `http://www.touchdevelop.com`.

[7] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *Proc. ECOOP*, pages 452–476, 2006.

[8] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. EuroSys*, pages 187–198, 2009.

[9] A. Cheung and S. Madden. Performance profiling with EndoScope, an acquisitional software monitoring framework. *PVLDB*, 1(1):42–53, 2008.

[10] 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. MobiSys*, pages 49–62, New York, NY, USA, 2010.

[11] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, May 2004.

[12] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. ASPLOS*, pages 279–292, 2013.

[13] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: code offload by migrating execution transparently. In *Proc. OSDI*, pages 93–106, 2012.

[14] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, 2003.

[15] I. Neamtiu and M. Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. PLDI*, pages 13–24, 2009.

[16] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: mobile app performance monitoring in the wild. In *Proc. OSDI*, pages 107–120, 2012.

[17] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: simplifying sensing and coordination tasks on smartphones. In *Proc. HotMobile*, 2012.

[18] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Madden, H. Balakrishnan, S. Toledo, and J. Eriksson. Vtrack: accurate, energy-aware road traffic delay estimation using mobile phones. In *Proc. SenSys*, pages 85–98, 2009.