# SplitSecond: Flexible Privilege Separation of Android Apps

Jehyun Lee
National University of Singapore
leejh@comp.nus.edu.sg

Akshaya Venkateswara Raja
Funding Societies, Singapore

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

***Abstract*—Android applications have been attractive targets to attackers due to the large number of users and the sensitive information they possess. After the success of the first step of an attack exploiting a software vulnerability, the consequential damage is primarily determined by the criticality and the amount of Android permissions that a victim application has. As a countermeasure, process separation techniques that isolate potentially vulnerable components — usually native libraries — from the critical data and permissions, have been proposed. However, existing techniques offer little flexibility in the separation, e.g., with all native code being placed into one process without considering its dependency with other (Java) components and the non-empty set of permissions needed. In this paper, we propose a flexible privilege separation system, named SplitSecond, that enables selective permission separation at the granularity of Java components and native methods. SplitSecond provides safety against the attacks by restricting permissions on a user selectable isolation unit. According to our case study and experimental evaluation on a real handset with SplitSecond adopted Android OS and 100 top-ranked Android applications, 59.59% of activities, 66.8% of native methods, and 47.49% of permissions on average are flexibly splittable by SplitSecond with moderate overhead.**

***Index Terms*—Android security, privilege separation, process isolation**

## I. Introduction

While Android becomes more popular as an operating system for mobile devices taking more than 80% of the market share [1], higher demands are placed on the privacy and security of its applications. This is partially due to the permission model used in Android in which once an application is compromised, the attacker takes over the control of not only the program codebase (to run an arbitrary sequence of instructions) but also the permissions assigned to the application (to perform sensitive operations or obtain sensitive data). Privilege separation and code isolation have been proposed as effective techniques to confine the damage created by a successful exploit to Android applications, typically by separating and isolating native code (widely considered as more vulnerable) [2]–[5] or advertising libraries (usually considered as untrusted) [6]–[10] into an unprivileged process.

However, existing solutions lack the flexibility in two dimensions, undermining their security and usability. First, existing solutions perform the separation and isolation at a coarse granularity, moving only the entire set of native or untrusted libraries to the isolated process. However, in reality, such potentially vulnerable and untrusted libraries might have tight data dependency with other parts of the application (e.g., an Activity and a Service), separating which would result in excessive inter-process communications or even loss of functionality. For example, in a simple Android application that we will use as a running example in this paper (more details are presented in Section II), the native libraries could not be isolated from its dependent Java component without substantial modification to the app. This data dependency problem on native code isolation had limited discussion in existing work, and the proposed solutions like binary rewriting to change the data flow require a thorough understanding of the target library [5], which could be unrealistic.

Moreover, untrusted Java code holding critical permissions could also pose a significant threat to privacy. Previous efforts in solving this problem propose fine-grained permission management with the component [11], the package [12], and the component transition context [13]; but they lack the attractive security property process separation provides in, e.g., completely blocking untrusted code from certain permissions.

Second, the set of permissions and privileges to be assigned to the isolated process is not necessarily empty. Afonso et al. experimented with a million applications and found that many native methods require permissions for Java method calls through JNI and system calls [14]. Coupling this with untrusted code, which is not necessarily limited to third-party advertising libraries but, e.g., methods or classes developed by a programmer from another development team in the same company, that may need certain permissions for its proper execution demands more flexible assignment of permissions to the isolated process. In the running example we are going to use in this paper, the Java code and the native libraries on the isolated process require `READ_STORAGE` and `NETWORK_STATE` permissions for connectivity check and accessing media files, without which these components simply do not work.

Creating a new process, moving fine-grained Java and native code, and assigning certain privileges to the isolated process may not sound overly complicated, but doing so in a reliable (without breaking the original semantics of the application) and automatic (without manual analysis) manner involves two main challenges. The first challenge is on recognizing data dependency among methods and classes in both Java and native code, which could make certain code pieces "non-splittable". For example, a static network port could not be trivially shared

between two processes with different UID, otherwise it could lead to access violation errors. Data dependency arises not only because of the accessibility and ownership of data but the validity of the identifier. Resource identifiers like file descriptors also introduce dependency through the file that is shared. In our running example to be discussed in more details in Section II, a Java Activity passes a file descriptor instead of a file path to native methods, and the identifier is only valid in the process where the file is opened. The second challenge arises specifically for proper execution of Java code on the newly created process, since specific runtime context (e.g., background tasks, global variables, and activities that had executed earlier) is needed for its proper execution.

In this paper, we present our design and implementation of a novel application execution system called SplitSecond, which provides a split and monitored execution environment that separates an application into two distinct processes. SplitSecond defines and recognizes *Split Execution Units* in an application with dependency analysis at the granularity of Android components and native methods, systematically organizes app permissions for flexible configuration of the split processes, and provides Android runtime components for the split execution. We apply SplitSecond to provide reliable splitting for the top 100 ranked applications on Google Play.

Besides showing that SplitSecond is capable of splitting Java and native code with the required permissions, we also evaluate the extent to which such flexible separation could be reliably performed without breaking application semantics. Through empirical case studies of the top 100 ranked applications running on a real handset with a modified Android OS adopting SplitSecond, we show that SplitSecond gains the flexibility of freely assigning, on average, 59.59% of activities, 66.8% of native methods, and 47.49% of permissions to either the main or the split process. According to our user-level overhead measurement with six sample applications, the separated execution raises 8 to 20% of overhead in time and less than 50% overhead in memory consumption.

## II. Motivation and Challenges

SplitSecond provides separated process execution of Android apps by creating a distinct process for split execution and managing native libraries, Android activities, and communications between a main and a split process. In the rest of the paper, we use the term *split* application and process to denote the additionally created app and the newly created process, while the original application and its process are referred to as the *main* application and process.

### A. Motivating example

We use the example of a media player application, GOM Player[1], available on Google Play to motivate our system and to explain our design and implementation. This media player supports playback of local video files as well as those downloaded from the web, and requires sensitive permissions including `INTERNET`, `CONTACTS` to access Google Drive[2], and `READ` and `WRITE_EXTERNAL_STORAGE` for media file management. It consists of 60 Android components including 45 activities and 6 native libraries.

*a) Potential threat:* The focus of most existing Android privilege separation studies [2]–[5] is the set of native libraries, which are usually coded with unsafe languages and are potentially vulnerable. Any successful exploitation into these native libraries could utilize all permissions given to the app to, e.g., leak sensitive information. In our example of the GOM player, these include six native libraries written in C/C++. The same argument goes to untrusted code which could intentionally compromise security and privacy of the application. Such untrusted code could be exploiting the permissions which are not really needed by itself but are nevertheless granted due to coarse-grained sharing of permissions with other legitimate pieces of code. The over-permitted and shared permissions could easily harm the users' privacy unintentionally due to the lack of privacy consideration in development and taking operation/development convenience by sacrificing user's privacy. This coarse-grained permission sharing problem of Android has been a common argument by previous studies [11]–[13] as one of the design level weaknesses.

*b) Complexity in separation:* Although GOM Player is a simple application with clear understanding of the potential threats, separating the native libraries into another process involves two main complexities. First, five out of the six native libraries performing video decoding and Digital Right Management (DRM) have tight data dependency with a Java component, `PlayerActivity`. `PlayerActivity` uses the five native libraries to provide video playback, and therefore is the best and only option to be moved to the split process together with the five native libraries because of the data dependency. Otherwise, de-coupled `PlayerActivity` and the dependent native libraries makes a fault when it tries to access the memory address of a shared object on the other process. However, existing solutions (e.g., NativeGuard [5]) do not support placing Java components onto the split process.

Second, the native libraries require certain privileges for their proper functioning. Specifically, video decoding needs `READ_STORAGE` permission, while the sixth library `libsqlcipher` needs `INTERNET` access. Existing solutions, however, assign zero permission to the split process.

*c) Targeted privilege separation:* Figure 1 shows an example of a targeted privilege separation. In this configuration, the split process executes `PlayerActivity` together with its closely coupled decoding and DRM native libraries, and is assigned a minimum set of privileges for its proper execution. The sixth native library, which requires `INTERNET` permission, runs in the main process.

Such a separation meets our security and privacy needs because the potential vulnerabilities in the five decoding and DRM libraries do not have `INTERNET` permission, greatly

[1]https://play.google.com/store/apps/details?id=com.gretech.gomplayerko&hl=en

[2]CONTACTS permission is the permission group to which GET_ACCOUNT permission (what the app actually needs) belongs.

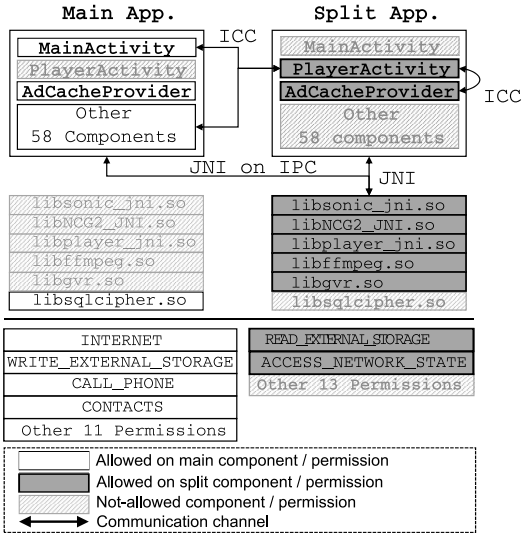


Fig. 1. Example of split execution of a media player. Split App and JNI-on-IPC are extensions by SplitSecond.

reducing the attacking surface and restricting private information from leaking to the Internet. At the same time, it helps solving the constraints in data dependency since the dependable components are in the same process.

We stress that our objective is not to achieve one specific configuration of privilege separation of Android application execution, but to support *flexible* ways of splitting. We want to analyze the data dependency in Android apps automatically, provide to end users *viable* separation with the notion of *Split Execution Unit (SEU)* (definition of which is introduced in Section III-A), and support runtime execution of the app in any specific configuration that the users deem suitable.

An example of such flexibility can be seen from the `AdCacheProvider` component. This component is special in that it is a local content provider used by activities in both the main and the split processes. SplitSecond supports the placement of such shared components on either process.

### B. Users of SplitSecond and assumptions

SplitSecond targets Android users who have a higher demand of system and application security. One practical usage model of SplitSecond is to provide a pool of pre-defined configurations to users to simplify the selection process. We assume that users do not understand code-level details of the target application because they are not the app developers, and therefore consider binary rewriting the application or changes to its implementation of data flow impractical.

### C. Challenges and our solutions

*1) Data and resource dependency:* Data and resource dependency arises due to the access control of Android and SELinux system which restricts accessibility to a resource in other processes with different UID/GID. Note that the read-only assets, such as media files and property tables carried by an app package (APK) are out of our concern because they are statically provided to both processes. Figure 2 summaries four cases in which the dependency of data and resources could make split execution of Android apps complicated.

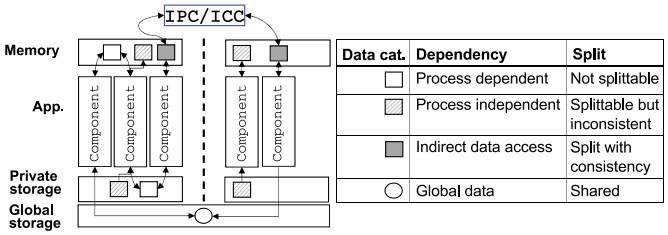


Fig. 2. Four types of data dependency

- Direct data access with a process dependent identifier: When the identifier of data is for direct access without any interface or a converter like a private memory address or a file descriptor, the identifier is invalid in any other processes. SplitSecond does not place components or native methods that share a process dependent identifier into two processes.
- Direct data access with a process independent identifier: This occurs in cases of a relative file path, a local resource path, and a relative URI. Because the identifiers are valid in each process, it does not immediately harm usability but could potentially result in data inconsistency.
- Indirect data access: When some native code attempts to read/write a Java object, the access occurs indirectly through JNI. Because the actual access to the data happens in the Java side of a JNI request, the customized JNI implementation could enables sharing of consistent data between two processes by converting an invalid identifier to a valid one in the process.
- Universally accessible data: In the example of GOM Player, media files in external or shared storage are globally accessible by both processes. SplitSecond, therefore, freely assigns corresponding components and native methods to the main or the split process.

*2) Execution environment for Java code:* A split process must have the required framework, libraries, and interfaces to other system layers similar to the original environment for its normal usability and necessary runtime context, including background tasks and initialization of global variables. SplitSecond obtains this runtime context by automatically executing the mandatory initialization code of an app when a split process is created.

## III. Design and Implementation of SplitSecond

Intuitively, SplitSecond achieves privilege separation by creating a new process from a distinct application with unique UID, memory area, permissions, cache directory, and files, which are protected by the Linux access control system. To further provide *flexibility* in its separation, SplitSecond is designed with the following two high-level principles.

First, SplitSecond uses a centralized native trampoline implemented in Android runtime to provide method-

level split execution for native methods, and an extended `ActivityManagerService` (AMS) to support split execution of Java components and control Inter-Component Communications (ICC) between the main and split processes. In other words, SplitSecond supports splitting at the granularity of native methods and Java components. Second, SplitSecond establishes and monitors JNI on IPC channels between the main and split processes to enforce specific privilege assignment to the Java and native methods.

In the rest of this section, we first discuss our static analysis for identifying dependency among native methods and Java components, and then briefly outline our strategy in privilege separation without violating data dependency. We then describe our design of the runtime system to support split execution and its implementation.

### A. Static analysis and Split Execution Units

Before presenting flexible privilege separation configurations to an end user, SplitSecond performs static analysis on the Android application to find permissions required by each native method and Java component as well as the data dependency among them. As discussed in Section II-C1, SplitSecond deals with four types of data dependency and performs potential splitting in different ways.

*a) Dependency among native methods:* Dependency on a native method and its caller components comes from the passing of a directly accessible resource reference from Java to native methods and between native method calls. When a Java method passes a resource reference which is accessible without JNI to a native method, the native method should be on the same process as the caller. To detect the dependent methods, we locate native method calls which have a byte buffer pointer, native resource pointers, or a file descriptor as an argument or return data type and mark them as "non-splittable" from their callers. In the example of GOM Player, three native libraries with JNI-exported native methods can not work in a distinct process from their caller component `PlayerActivity` because their argument includes a file descriptor which is a process dependent resource identifier. SplitSecond therefore considers them "non-splittable".

*b) Dependency among Android components:* In contrast to native depending on a single object (which is its caller), an Android component has dependency with multiple components even including its callee native methods. As a logical unit of the dependent components, we introduce the concept of *Split Execution Unit (SEU)*. An SEU is a closed set of one or more Android component classes among which there is data dependency and execution has to happen in the same process. Note that once defined, an SEU has its implication at the code level (that to be executed in the same process) and permission level (that be given some specific permissions). If a component requires prior existence of another component, we group both into the same SEU and denote the one with prior existence as an anchor component. The anchor components of an SEU are either 1) activities and services that can be started without prior components; or 2) parents of sub- or child-activities.

Intuitively, an SEU consists of anchor components and components connected with data or runtime context dependency. SplitSecond performs the following four steps of processing on the disassembled SMALI [15] code in order to identify SEUs and their minimum set of permissions required.

1) Processing the manifest: We locate activities and services declared in `AndroidManifest.xml` and group those with parent-child and sequential relationship into the same SEU.
2) Tracking control flow: We find all possible control-flow paths in the method-level granularity. While the variables accessed on the control-flow paths are used for finding dependent classes and components, the methods on the control-flow paths indicate permissions potentially required.
3) Finding dependencies: We group components with runtime context and/or data dependency into the same SEU. In a conservative manner, we detect components that access the same member variables in a class.
4) Tracking permissions: When any control-flow paths of an SEU reach an API call that requires an Android permission, we consider the corresponding permission required by the SEU.

Eventually, a set of SEUs are identified for the target application, from which an end user can formulate a specific assignment of SEUs to the main and split processes. Note that the number of SEUs obtained and their corresponding sizes are good indicators on the amount of dependencies in the app as well as the amount of code-level flexibility the end users have in configuring SplitSecond. We further evaluate this flexibility on the top 100-ranked Android applications; see Section IV-A.

### B. Flexibility in permissions assignment

Besides selecting the SEUs to be executed on the main and split processes, another important configuration for SplitSecond is the corresponding permissions to be assigned. This might sound trivial since our identification of SEU discussed above also outputs the set of permissions required by each SEU, and intuitively, one could simply assign the union of the corresponding permissions to the main and split process.

However, SplitSecond provides further flexibility (and the corresponding security property) by defining a third category of permissions. When a native method in a split process requests a Java method call through JNI, SplitSecond could forward the JNI call to either the main or the split process, provided that the corresponding permission is given to the process chosen. We, therefore, have the following three sets of permissions defined.

- $\mathsf{P}^{\mathsf{M}}$: Permissions granted to the main process
- $\mathsf{P}^{\mathsf{S}}$: Permissions granted to the split process
- $\mathsf{P}^{\mathsf{S}\to\mathsf{M}}$: Permissions granted to Java methods in the main process that serve JNI requests from the split process

A naive way of performing permission assignment is to let $\mathsf{P}^{\mathsf{M}}$ be a superset of $\mathsf{P}^{\mathsf{S}\to\mathsf{M}}$. However, SplitSecond provides additional support on $\mathsf{P}^{\mathsf{S}\to\mathsf{M}}$ to provide more flexibility in

the permission assignment and as a result, enhanced security properties. The reason is that by making $P^{S \to M}$ an isolated set of permissions (excluded from $P^M$), SplitSecond can selectively grant permissions to such JNI requests from the split process (which is potentially vulnerable or untrusted) by allowing only a subset of native caller methods and a subset of Java callee methods.

### C. User configuration and app installation

With the two sets of analysis results (code-level analysis as discussed in Section III-A and permission-level analysis as discussed in Section III-B), SplitSecond presents a set of required permissions according to chosen activities or native methods by the user, and a set of SEUs according to any permissions selected by the user. We call the former configuration "code-driven" and the latter "permission-driven".

In the code-driven configuration, a user has a set of anchor components and the splittable native methods as a pool of selection. Once a component or a method is chosen to execute on the split process, the set of dependent components, methods, and permissions (SEU) will follow to the split process, too. On the other hand, if a user selects a permission, SEUs requiring the permissions are sent to the split process, and the anchor components and the splittable native methods belonging to the SEUs are enlisted to the permission-driven split configuration.

A final configuration consists of permissions for $P^M$ and $P^S$ which will be applied to the main and split processes respectively, a list of split activities, a list of split native methods, and lastly, the list of Java methods to be accepted or rejected for $P^{S \to M}$. SplitSecond then loads this confirmation during application installation and execution for proper settings.

The installer of SplitSecond automatically installs two distinct applications during the initial installation of an app so that the split application has its unique UID for proper protection.

### D. Runtime components

In this section, we introduce the detailed components of SplitSecond runtime based on `AOSP_7.1.0_r36` [16], see Figure 3. The gray colored components in are newly introduced by SplitSecond, while the dotted ones indicate extended components. SplitSecond uses the main process to handle the split configuration, and the split process acts as a server to take split executions. The split process management service aids split process and permission management whereas the intent handler, native invocation handler, proxy JNI, and object reference manager help communication between the main and the split processes.

*1) Split process management service:* The split process management service helps the native invocation handler to find the binder interface of the corresponding split application, and requests to start the split process to the `ActivityManagerService (AMS)` when the split process does not exist. The newly started split process automatically loads the Proxy JNI library for working as a binder server. Once the main process receives the binder interface from the service, the main process communicates with the split process directly without this service.

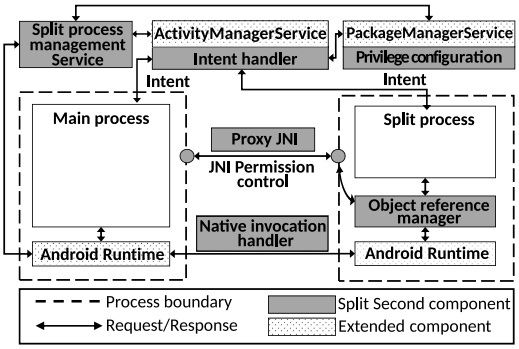


Fig. 3. Runtime components of SplitSecond and their communications

*2) Intent handler:* Intent handler is the main controller of Java code splitting. The intent handler implements the Android component separation by extending AMS that manages the intent transactions between Android components. The intent handler catches an intent transaction and modifies the package name from that of the main process to that of the split process (or vice versa) along with the split configuration.

*3) Object reference manager:* Because the main and the split processes have their protected memory and address spaces, the variables in a process are not accessible from the other unless a user places them on shared memory area in the kernel. Fortunately, primitive data type variables and arrays of primitive data types are transferable through the binder in both ways. However, all the other variable types require marshaling to be sent through the binder interface. Even though several object types could be marshalized, we cannot guarantee the validity of the transferred object in the receiver processes. SplitSecond solves this object synchronization problem by transferring object identifiers and actions toward the objects instead of object bodies. In Android, the object identifier transferred through JNI is an indirect object reference (identifiers pointing at Java objects indirectly).

*4) Native invocation handler:* To invoke a native method on the split process, SplitSecond sends the invocation request to the split process through the binder. The native invocation handler performs a transaction in three steps: native method call catch, configuration check, and binder transfer for native library loading and native method invocation. Figure 4 illustrates a general process flow of the native invocation handler and proxy JNI server/client.

*5) Proxy JNI server and client:* Proxy JNI is the implementation of binder interfaces for JNI on IPC. The Proxy JNI server and client have identical interface methods to JNI implementation and send requests and returns to the paired processes through binder interfaces. These proxy interfaces support Java method calls and object access from a native method between the main and the split processes. When a native method is in the split process but the parameter objects and required permissions are on the main process, the Proxy client on the split process passes the JNI request to JNI server on the main process. The server-side binder interface makes a local JNI call and returns the result through the binder again.

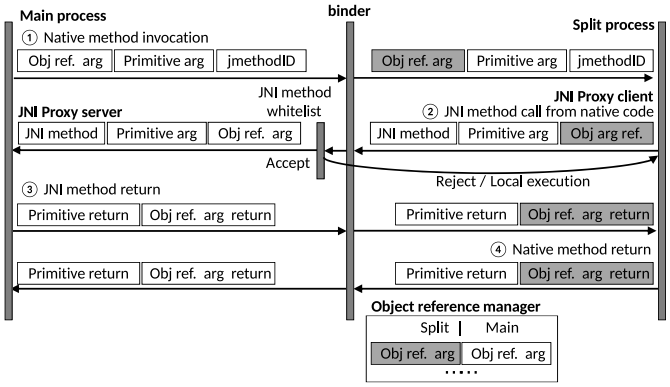


Fig. 4. Native method invocation and privilege enforcing

Proxy JNI also manages the permissions in $P^{S \to M}$; see Section III-B. When a native method in a split process requests for a Java method call, the JNI server checks the user configuration before allowing it or denying it. JNI client passes the JNI call in local JNI environment when an error is returned. If the split process does not have the necessary permission either, the action is blocked and is handled as cases of permission denial.

## IV. Experimental Evaluation

In this section, we present our evaluation results in using SplitSecond for the top 100 ranked Android applications on Google Play. Our evaluation focuses on the flexibility SplitSecond provides in code-level and permission-level configurations as well as the resulting overhead. Besides the running example of GOM Player, our evaluation will also focus on five high-profile applications with large codebases. Note that we do not focus on the security analysis here since it is naturally provided by Android when execution is within two different processes. Table I shows some details of these six applications.

Runtime experiments were conducted on a Google Pixel phone with a 64-bit ARM processor, 32 GB storage, 4 GB RAM, and our modified OS based on AOSP 7.1.2_r36 [16].

### A. Flexibility: how much code and permission can be moved to the split process

SplitSecond is designed to provide flexibility in privilege separation. As we discussed in Section III, such flexibility can be measured at the code level and permission level: at the code level, a user may prefer moving certain potentially vulnerable or untrusted code to the split process, while at the permission level, a user may prefer assigning a small subset of the privileges to the split process. However, this is constrained by the data dependency among Java components and native methods, as well as the necessary execution environment to be set up for Java execution. To evaluate the extent to which SplitSecond provides flexibility to users, we apply SplitSecond on the top 100-ranked Android applications on Google Play and measure the percentage of activities, native methods, and permissions that can be split into a separate process. Result shows that overall, 59.59% of activities, 66.8% of native methods, and 47.49% of the permissions can be removed from the main process.

Although this initial result is encouraging in the sense that SplitSecond provides flexibility in configuring about 60% of code and 50% of permissions, code-level and permission-level flexibilities are highly coupled in enforcing certain security and privacy policies, and they have to be cross-analyzed for more meaningful evaluation. For example, splitting more vulnerable code without or with limited permissions give higher security to a user. That said, if a permission is removable from the main process by splitting a lot of (potentially vulnerable and untrusted) code to a split process without harming normal usage, the main process can be safer from unwanted/unexpected permission exploitation.

For this purpose, we perform a more detailed analysis on five high-profile applications together with our running example GOM player, and present the results in Table II. We discuss the detailed results in the rest of this subsection.

*1) How much Java code can be split:* Intuitively, the more code that can be split into a separate process, the more flexibility a user has in enforcing various security and privacy policies. As we discussed in Section III-A, the smallest unit in splitting Java code is a Split Execution Unit (SEU). Smaller SEUs (and consequently larger number of SEUs) imply less dependency among the Java code and more flexibility in moving them around.

The first three rows of Table II show that there are 20 SEUs for simple apps (e.g., GOM Player) and as many as 110 SEUs for complex apps (e.g., Twitter). The average number of activities per SEU is between 1.63 and 3.41. Not shown in Table II, we calculated $c = 1.77$ on average for the top 100-ranked apps. This shows that SplitSecond achieves good flexibility among Java code with large numbers of SEUs to be moved between the main and split processes, and the dependency among Java activities are small with, on average, only one or two activities being tightly coupled. We believe that this is a result of event-driven programming for Android.

The results also suggest that privilege separation in an Android application could potentially go a long way with, say, more than 10 isolated processes. It is not clear, though, if there are actually practical security needs to go to that extent.

*2) More Java code split $\Longrightarrow$ More permissions?:* We have shown that SplitSecond provides great flexibility in terms of the number of (independent) SEUs to be executed on the split process. Recall that SplitSecond provides flexibility at the permission level, too, in that selected permissions could be given to the split process. We now go deeper into the analysis of the number of permissions to be assigned when more SEUs are moved to the split process.

Our analysis here assumes that the user intends to assign the largest amount of code to the split process with the minimum number of permissions. Figure 5 shows the possible configurations the user could choose from for the six Android applications. When no permission is given to the split process (first two bars), one can assign between 21.82% and 65.00% of the SEUs to the split process, which is consistent with the

TABLE I
PROPERTIES OF SIX SAMPLE ANDROID APPS IN OUR EVALUATION

| App name | Version | Size of APK | # of Java classes | # of activities | # of native lib. | # of native methods |
|---|---|---|---|---|---|---|
| GOM Player | 1.4.2 | 34 MB | 7,222 | 45 | 6 | 192 |
| Twitter | 7.33.1 | 36 MB | 29,212 | 192 | 10 | 141 |
| Uber | 4.197.10002 | 20 MB | 21,145 | 235 | 11 | 189 |
| Carousell | 2.51.6.20 | 27 MB | 18,539 | 129 | 13 | 176 |
| Lazada | 6.3.1 | 22 MB | 19,891 | 90 | 24 | 102 |
| Pinterest | 7.0.0 | 22 MB | 21,832 | 29 | 16 | 559 |

TABLE II
FLEXIBILITY ON SPLIT EXECUTION FOR FIVE SAMPLE APPS

| | GOM Player | Twitter | Uber | Carousell | Lazada | Pinterest |
|---|---|---|---|---|---|---|
| # of activities (***a***) | 45 | 192 | 235 | 129 | 90 | 29 |
| # of SEUs (***b***) | 20 | 110 | 69 | 79 | 45 | 21 |
| # of activities per SEU (***c***=*a*/*b*) | 2.25 | 1.75 | 3.41 | 1.63 | 2 | 1.38 |
| # of SEUs not requiring permissions (***d***) | 13 | 24 | 43 | 26 | 13 | 4 |
| # of exported native methods (***e***) | 192 | 141 | 189 | 176 | 102 | 559 |
| # of exported native methods that can be de-coupled (***f***) | 180 | 72 | 68 | 69 | 102 | 93 |
| % of exported native methods that can be de-coupled (***g***=*f*/*e*) | 93.75% | 51.06% | 35.98% | 39.20% | 100% | 16.63% |
| # of ***f*** while requiring permissions (***h***) | 2 | 0 | 10 | 0 | 1 | 0 |
| # of permissions required by the app (***i***) | 15 | 33 | 31 | 20 | 36 | 18 |
| # of permissions required by SEUs (***j***) | 6 | 17 | 19 | 11 | 14 | 6 |

values of ***d*** shown in Table II. In addition to that, assigning two permissions to the split process will allow more than 50% of the SEUs to be moved to the split process, which could be a sweet spot in user's configuration (gaining substantial security benefits of separation while assigning minimal permissions to the split process).

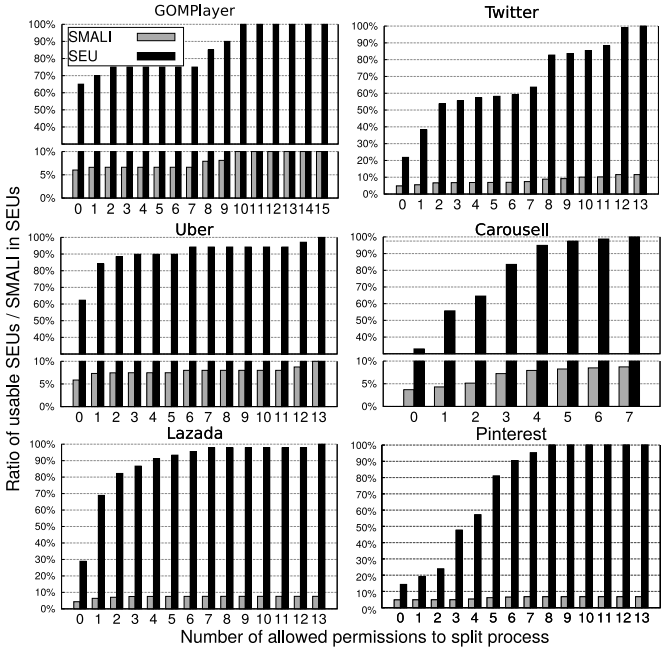


Fig. 5. SEUs and SMALI code size at split process with permissions given

Figure 5 also shows the amount of SMALI instructions executed on the split process. An interesting observation here is that the rate of increase for the percentage of SMALI code is significantly lower than that for the SEUs, which suggests that SEUs requiring permissions typically contain smaller code size. Another way to look at this is that code involving Java permissions exhibits less dependency on data and runtime context. This means that the initial gain in security with none or a small number of permissions given to the split process is relatively big (with a large amount of code running on the split process), while the marginal security gain (in terms of additional code running on the split process) goes down with more permissions given to the split process. Note that the percentage of SMALI code is far from reaching 100% even when all SEUs are running on the split process due to non-movable Java code (for execution environment setup), non-invoked library modules, data structure classes, background task classes that are not services, etc.

We extend the analysis to the top 100 ranked apps. The SEUs of the top-ranked apps require 7.85 of permissions on average out of 13.43 permissions required by an app. The pool of flexible permission separation, therefore, is near half of the permissions used by an app. In the splittable component perspective, the top-ranked apps had 24.95 SEUs (59.59% of activities), and 16.18 SEUs of them required one or more permissions on average. The remaining 8.77 SEUs (20.95% of activities) did not require any permissions. Regarding permission-to-usability trade-off, 20.95% of the activities can be separated into the split process without any permissions, and a user can get a much better ratio by allowing a few common permissions. Lastly, the removable permissions from the main process by splitting every possible Java and native code is 6.38 (47.49% of the permissions required by an app).

*3) De-coupling native methods:* The analysis above shows the flexibility SplitSecond provides with Java code along with the corresponding permissions to be given to the split process.

Figure 5 shows that users may prefer having fewer SEUs at the split process in order to minimize the permissions given. In the following analysis, we investigate the possibility of de-coupling native methods from their corresponding parent SEUs so that the native methods can be executed on the split process while the parent SEU is tied to the main process. Here, our focus is on a subset of JNI-exported native methods which do not have any data dependency with its caller Java method. Non-JNI-exported methods cannot be de-coupled because they can only be called by the other native methods.

We find that the percentage of such a subset of JNI-exported native methods that can be de-coupled varies from 35.98% to 100% (see values of $\boldsymbol{g}$ in Table II) from the six sample apps. This percentage is on average 66.8% on the top-100 ranked Android apps. Our manual analysis shows that methods cannot be de-coupled mainly due to the use of direct memory access for efficient memory copying and image processing. Interestingly, most of the methods that can be de-coupled do not require any permissions to execute (see values of $\boldsymbol{h}$ in Table II), which makes such de-coupling really attractive.

SplitSecond also provides flexibility in handling the native methods that require permissions. In the case of Uber, ten native methods require six permissions for sensitive JNI calls but not for GID-managed object. Therefore, regarding permission categorization, a reasonable configuration for Uber is to keep $\mathsf{P}^{\mathsf{S}}$ empty and let $\mathsf{P}^{\mathsf{S}\rightarrow\mathsf{M}}$ contain the six permissions. One can also move the relevant permissions from $\mathsf{P}^{\mathsf{S}\rightarrow\mathsf{M}}$ to $\mathsf{P}^{\mathsf{S}}$, or to place the ten native methods in the main process.

### B. Overhead

Overhead of SplitSecond comes with the following main contributors.

- Loss of optimization due to the centralized and serialized point for native method trampoline and permission enforcement.
- Additional process creation and execution, as well as the extended Android components to catch native methods and intents.
- Additional context switches between the main and the split processes.
- Additional binder transactions for every native method call, JNI call, and return of results. This will likely introduce negligible overhead, though, as only references and primitive type arguments and return values are transferred.

We evaluate the overhead of SplitSecond experienced by end users in term of memory and time with the five high-profile applications and the simple media player application.

We set two different usage scenarios, i.e., app initialization with a split configuration which de-couples all the exported native methods without Java code, and app specific usage scenarios which include Java code separation or less dense native method de-coupling, and measure the time and memory overhead. The app initialization scenario is tested from the start of an app to the display-complete message of the last activity without any user input. The app specific usage scenarios are posting a tweet for Twitter, opening a PayPal payment activity for Uber, opening an image gallery activity for Carousell, searching products on a webview for Lazada, and opening picture upload activity for Pinterest. Within the app specific scenarios, Twitter and Lazada have only native method separation during the test because those actions are conducted on a view without change of activity.

All the scenarios are started by sending an intent. We tested each initialization scenario for one hundred times on our handset and measured the average elapsed time using an Android Debugging Bridge (ADB) command sending intents repeatedly with an option of killing and restarting the existing process instead of waking up the process. Similarly, we also tested the app specific usage scenarios for one hundred times in the same environment. The elapsed time is measured from sending an intent toward a target activity to the completion of display reported by `ActivityManager` and `WindowsManager`. The memory overhead is the difference of average memory usage between the stock OS and SplitSecond. The average memory usage is measured by using *top* command through ADB during each app specific usage scenario, and we take the stable values right after completion of the given work.

Table III shows the result of the percentage of additional time and memory SplitSecond introduces in comparison with running the same operations on the stock Android OS.

*a) Runtime overhead for app initialization:* The first column of Table III shows the time overheads of each app during initialization. We set a split configuration to decouple all the possible native method calls into the split process. The time overhead includes process creation and initialization time for the split process, IPC transactions from the main process to the split process for the decoupled native method calls, and backward JNI-on-IPC transactions from the split process. According to our observation, the differences among the applications are caused by loading of mandatory native libraries and creating background threads in the split process. For example, debugging and exception handling native libraries on GOM Player and Lazada are commonly loaded by the main and split process, which makes it long for the creation of a split process. In contrast, the effect of IPC transactions was not a critical factor. GOM Player showed only two JNI-on-IPC transactions without main-to-split native method calls, whereas Pinterest made one native method call and nineteen JNI-on-IPC transactions with thousands bytes of parcels.

*b) Runtime overhead for app specific usage:* We set application specific usage scenarios for each app including Java-code separation. Only Twitter and Lazada have the configurations that separate native methods. Note that these scenarios do not include process creation and app initialization time for the split process. Therefore, the time overhead on the second column of Table III represent ICC handling, split policy comparison, and GUI level context switching overhead. As we can find in the cases of Lazada and Twitter, the native method separation which does not make a change of the foreground application showed relatively small time overhead.

In summary, we find that a native-only separation in the

TABLE III
OVERHEAD IN TIME AND MEMORY IN TWO TEST SCENARIOS

| App name | Time (app init.) | Time (app specific) | Mem. |
|---|---|---|---|
| GOM Player | 38.58% | 22.49% | 58.24% |
| Twitter | 12.21% | 11.47% | 51.32% |
| Uber | 17.52% | 12.57% | 42.84% |
| Carousell | 21.88% | 24.78% | 36.29% |
| Lazada | 37.26% | 11.56% | 27.36% |
| Pinterest | 17.53% | 18.07% | 43.83% |

initialization scenario and Java-code separation scenario show 12% to 38% and 11% to 24% of runtime computation overhead, respectively, which is diverse and dependent on the application characteristic. Memory consumption presents a persistent overhead by additional process and is mainly dependent on the size of the application code base and size of the split activity.

## V. LIMITATIONS AND FUTURE WORK

Our prototype implementation of SplitSecond shows the feasibility of flexible privilege separation. Here we summarize limitations of SplitSecond and our future work.

*a) Permissions required by shared components:* Our experiments show that apps with background services and tasks have limited flexibility in code splitting under SplitSecond. Most of the real-world applications use background tasks for asynchronous processing of heavy or periodic work, and the tasks are commonly shared among multiple activities. Therefore, the code of the shared components and the required permissions are likely used by both the main and the split processes. We envision using a message handler to give flexibility to such components.

*b) Attacks on ICC and IPC:* SplitSecond makes use of the distinct UID for protection, but attackers could target ICC or IPC instead. Though only the pre-determined JNI-on-IPC calls are allowed by the receiver process, the set of available objects and methods can be powerful enough to achieve a critical malicious action, and distinguishing exfiltration from normal communications is non-trivial. Context-aware malicious ICC detection [13] can be a complementary solution against these attacks.

## VI. RELATED WORK

Many previous efforts have been made to protect Android privileges from being abused or exploited. Because the privilege holder in Android systems is a User ID (UID), a main stream of previous approaches was UID/process separation. Once the code is separated to an isolated process, the privileges are protected by the Linux kernel. On the other hand, several studies attempted to manage the critical resources, including memory area, system calls, and API calls, by themselves without isolating the processes.

*a) Privilege control with an isolated process:* As briefly discussed above, placing a set of code in an isolated process having a distinct UID gives kernel level protection unless an attacker hijacks a higher privileged UID. Isolating native code from a Java application into an isolated process has been proposed, such as Robusta [3], [4] and *NaCl* [2], because native code is more likely to contain exploitable vulnerabilities.

The effectiveness of native code isolation is enhanced in Android systems due to the introduction of permissions. A study by Afonso et al. [14] provides more insight about the native libraries that are the major isolation targets in recent Android applications. According to their experiments with a million apps, placing the native libraries with no permission is not ideal because the native methods also require the permissions for Java method calls through JNI and system calls. This motivates our project on more flexible separation of Android app components.

One way to separate execution for a specific set of code is app repackaging. In particular, NativeGuard [5] proposed by Sun et al. is one of the state-of-the-art studies in process isolation approaches, and it shares some common motivations with our design. NativeGuard separates an app package into two sub packages, one for Java code and the other for native code. On the other hand, Dr. Android and Mr. Hide [17] implement native code isolation by injecting a broker module and rewriting system libraries. The broker based handling provides robustness against integrity check in contrast to app repackaging. In spite of clear isolation of the native code, assumptions on native code isolation approaches face limitations. As shown by Afonso et al. [14], several permissions should be given to the native methods for proper execution of most applications. In comparison with previous native code isolation approaches, SplitSecond provides flexible code separation and enhanced handling for the permissions without rewriting or repackaging of an app.

*b) Privilege control without isolated processes:* The key advantage of access control without isolated process is to avoid system overhead caused by the additional process. Compac [12] provides component-level access control by monitoring every component transition. The policy manager of Compac revokes and re-grants the permissions of an app by a component transition time. However, a reflection call allows Java and native codes to call a Java method in a different class without component transition, and this is a common drawback of Java level permission control approaches. Aurasium [11] inserts security policy checking code into the Java code and native libraries by repackaging an application. These app repackaging approaches commonly have a shortcoming against integrity check to detect the maliciously modified app packages. CoDRA [18] and APex [19] provide an extended policy enforcement framework for fine-grained permission management, even in an individual resource access level. FineDroid [13] provides a context-aware permission control distinguishing abnormal intra and inter-application contexts against unexpected exploitation of a granted permission. In contrast to isolation-based approaches, these fine-grained permission enforcement only block privileged actions but do not support "privileged but isolated execution" which is usually more attractive. As a hardware-supported approach, FlexDroid [20] provides strong data and privilege security with

memory isolation, but the coverage of configurable code is still limited to native libraries, though it also provides flexible permission configuration.

*c) Protection with faked environments:* The protection against privileged actions can also be achieved by providing worthlessness data or faked functionalities. MockDroid [21], AppFence [22], and TISSA [23] commonly proposed a privacy and critical resource protection method by providing faked or anonymized data and resources, like a fake network interface, instead of blocking privileged actions. Meanwhile the fine-grained permission systems likely crash against the suspicious actions, the isolated or faked environments provide more usability. Compared to an isolated process which is an indistinguishable environment without capability of the system-wide view, a mocked environment has an issue on detectability of the mocked data and interfaces.

## VII. Conclusion

SplitSecond is a flexible privilege separation system that provides component-level privilege management. We present the design and implementation of SplitSecond, overcoming a number of implementation challenges including data dependency and execution environments, and demonstrate the success in applying SplitSecond to the top 100 ranked Android applications. We also evaluate the extent to which SplitSecond can provide flexibility for users to configure the amount of code and permissions to be assigned to the split process, and show that SplitSecond gives flexibility to about 60% of code and 50% of permissions with moderate overhead.

## References


[1] Statista Inc., "Global mobile OS market share," 2019, https://www.statista.com/statistics/266136/.

[2] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[3] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the native beast of the jvm," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 201–211.

[4] M. Sun and G. Tan, "Jvm-portable sandboxing of javas native libraries," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 842–858.

[5] ——, "Nativeguard: Protecting android applications from third-party native libraries," in *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. ACM, 2014, pp. 165–176.

[6] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. Acm, 2012, pp. 71–72.

[7] S. Shekhar, M. Dietz, and D. S. Wallach, "Adsplit: Separating smartphone advertising from applications." in *USENIX Security Symposium*, vol. 2012, 2012.

[8] X. Zhang, A. Ahlawat, and W. Du, "Aframe: Isolating advertisements from mobile applications in android," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 9–18.

[9] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 89–103.

[10] J. Huang, O. Schranz, S. Bugiel, and M. Backes, "The art of app compartmentalization: Compiler-based library privilege separation on stock android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1037–1049.

[11] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium: Practical policy enforcement for android applications." in *USENIX Security Symposium*, vol. 2012, 2012.

[12] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, "Compac: Enforce component-level access control in android," in *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. ACM, 2014, pp. 25–36.

[13] Y. Zhang, M. Yang, G. Gu, and H. Chen, "Finedroid: Enforcing permissions with system-wide application execution context," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2015, pp. 3–22.

[14] V. M. Afonso, P. L. de Geus, A. Bianchi, Y. Fratantonio, C. Kruegel, G. Vigna, A. Doupé, and M. Polino, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy." in *NDSS*, 2016.

[15] B. Gruver, "Smali: An assembler/disassembler for androids dex format," 2019, https://github.com/JesusFreke/smali.

[16] Google Inc., "Android Open Source Project," 2018, https://source.android.com/.

[17] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: fine-grained permissions in android applications," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 3–14.

[18] N. K. Thanigaivelan, E. Nigussie, A. Hakkala, S. Virtanen, and J. Isoaho, "Codra: Context-based dynamically reconfigurable access control system for android," *Journal of Network and Computer Applications*, vol. 101, pp. 1–17, 2018.

[19] M. Nauman, S. Khan, and X. Zhang, "Apex: extending android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM symposium on information, computer and communications security*. ACM, 2010, pp. 328–332.

[20] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, "FLEXDROID: Enforcing In-App Privilege Separation in Android," in *NDSS*, 2016.

[21] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th workshop on mobile computing systems and applications*. ACM, 2011, pp. 49–54.

[22] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 639–652.

[23] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh, "Taming information-stealing smartphone applications (on android)," in *International conference on Trust and trustworthy computing*. Springer, 2011, pp. 93–107.