A Context-Aware Kernel IPC Firewall for Android

David Wu
davidxiaohanwu@gmail.com

Sergey Bratus
sergey@cs.dartmouth.edu

Department of Computer Science
Dartmouth College

ABSTRACT

Our phones go wherever we go. Ever present, and with ever more data and connections, smartphones hold as much sensitive data as traditional systems but do not have the same protections. Android's recent 6.0 (Marshmallow) release introduced much needed dynamic permission checks for applications. However, this does not go far enough in adapting to mobile phone's unique security needs. Smartphones encounter a wide variety of settings and situations that current security solutions fail to account for. We introduce a context-aware IPC firewall for Android that dynamically filters messages based on environmental data. Our `BinderFilter` can both block and modify Android IPC messages sent through Binder, which is in a position of complete mediation in Android. Our Binder hooking framework and message parser are unique in their scope and implementation—and mitigate broad classes of cross-app attacks, such as "collusion" and "UI-based activity hijacking" attacks. We also provide a policy application, `Picky`, with which users can set policy rules for any message and target applications. `BinderFilter` and `Picky` are free software, available at [1, 2].

1. INTRODUCTION

Android is the world's most popular mobile operating system with over 1.4 billion users [3]. The wide variety of sensor and personal data that phones store make them rich targets for attackers and data miners masquerading as third-party applications. Recently, various health tracking applications were found to export medical information to third-party servers [4], and Facebook's Messenger application has faced criticism for requiring extraneous permissions [5].

Android's security architecture is based on permissions [14]. Applications request permissions to access various system APIs such as Camera, Location, Microphone, and user data such as Contacts, Calendar, and External Storage. In October 2015, Google released version 6.0 of Android, named Marshmallow, which included major security updates such as dynamic permission checking for certain dangerous permissions [6]. Prior to Marshmallow, applications would statically request permissions at install time, and users would have no way of revoking or changing application permissions. Dynamic permission checking in Android Marshmallow filled a significant need in Android security policy. However, it is not sufficiently granular, nor does it expose all permission decisions to users. Furthermore, as Marshmallow has been adapted by only 7.5% of users as of May 2016 [7], additional security architectures are needed to ensure user privacy for all.

Android phones suffer from malware somewhat disproportionately. At the core of the problem is the fact that malware can commandeer other apps to do its its dirty work, abusing the technology that Android provides for legitimate communications between apps. Since interaction between apps is desirable and part of the normal functionality, it is not easy to close that malware attack vector, despite some new security improvements recently released.

In a sense, Android's app microcosm problem is the same as the Internet macrocosm's: we *want* computers and apps to talk to each other; we just want to be able to control who's talking to whom. Just like the Internet has developed the mitigation of the firewall to block undesirable communications, so the Android platform has been developing a means for blocking selected bad inter-app communications.

Just as firewalls evolved, the depth of examination of individual messages increased, and concepts of state were added, similar evolution is taking place on the Android platform. Our design, presented in this paper, aims to provide both the deepest level of message examination and the richest context for policy decisions.

While most security add-ons for Android see the need for some type of dynamic, fine-grained blocking, only a handful such as [29, 30] have focused on the *mobile* nature of mobile phones. Our approach focuses on the current specific needs of Android security by designing and implementing the following.

- *A novel context-aware security system for Android* that includes sensor, network, and system state data into policy decisions.
- *A message hooking framework for Android's Binder IPC system* that provides the necessary feature of complete mediation of Android IPC messages, including intents and permissions.
- *Blocking and modifying message data*, including more

DOI: 10.475/123_4

dynamic permissions than Android Marshmallow exposes to users.

- *A Binder message parser and formatter*, allowing for dynamic analysis of every Android IPC message.
- *A user application that abstracts security policy* and allows dynamic message blocking of any message and application.

Table 1 shows how our project improves on existing Android security add-ons.

We survey the related work in section 6. Briefly, other hooking frameworks for Android such as Heuser et al.'s Android Security Modules [11] and rovo89's Xposed Framework [12] have not demonstrated complete mediation of all IPC messages. Ours is the first of its kind that hooks *Binder*. Nitay Artenstein and Idan Revivo demonstrated Binder message parsing and data exfiltration but did not go as far as systematically hooking and modifying message data [13].

The remainder of this paper is as follows. We first review the background on Android's Binder IPC in section 2, then discuss our IPC firewall design considerations and implementation in section 3. Section 4 gives a brief overview of our UI application for configuring the firewall. Section 5 evaluates the mitigations our firewall provides against broad classes of attacks and malware abuses of the Android platform, and section 6 surveys previous work.

2. BACKGROUND

The Android platform is a set of libraries, compilation tools, interpreters, and kernel drivers built on Linux. It inherits Linux's file permissions, system call architecture, and, in Android versions after 4.3, SELinux policies. Android application permissions correspond to capabilities and are enforced by the PackageManager service via `checkPermission`. System permissions (such as installing packages or rebooting) are only available to system services and applications such as Settings and Google Play Store located in `/system/app` or `/system/priv-app`, which are read-only enforced by Linux. Network sockets and writes to external storage are also enforced by the kernel via Linux group ID's (GIDs). Each application is given a unique Linux user ID (UID) on installation. Android then enforces application memory sandboxing by running each application as its own process under its unique UID. File access is restricted by the underlying filesystem and SELinux policy, whereas service and RPC function calls are restricted by Android's PackageManager based on UID.

2.1 Binder

Communication between sandboxed applications is done via Binder inter-process communication (IPC). Binder replaces Linux's own IPC system in Android and enables uniquely identifying security tokens, death notifications, and (intra-package) RPC. Intents, Messengers, and ContentProviders are all built on Binder. (Intents in Android are asynchronous messages passed between applications to request data or to start a new activity.)

2.1.1 Binder Driver

Binder is implemented as a Linux kernel driver (`/dev/binder`), which is exposed to userland processes using the `ioctl()` syscall. The Binder driver is also responsible for copying data between sandboxed user processes, including data buffers, file descriptors, and death notifications.

The Binder driver `ioctl()` call takes as a parameter `binder_write_read`, which contains information about driver buffer consumption and pointers to marshaled user transaction data. The `write_buffer` and `read_buffer` fields point to `binder_transaction_data` objects. Those contain sender pid, receiver pid, uid information, and pointers to data buffers and offsets. Specifically, the `data.ptr.offsets` field points to `flat_binder_object` objects. Finally, `flat_binder_object` contains extra information such as file descriptors. These structures are listed below.

```
struct binder_write_read {
        signed long      write_size;
        signed long      write_consumed;
        unsigned long  write_buffer;
        signed long      read_size;
        signed long      read_consumed;
        unsigned long  read_buffer;
};

struct binder_transaction_data {
        union {
                size_t handle;
                void *ptr;
        } target;

        void *cookie;
        unsigned int code;
        unsigned int flags;
        pid_t sender_pid;
        uid_t sender_euid;
        size_t data_size;
        size_t offsets_size;

        union {
                struct {
                        const void *buffer;
                        const void *offsets;
                } ptr;
                uint8_t buf[8];
        } data;
};

struct flat_binder_object {
        unsigned long type;
        unsigned long flags;
        union {
                void *binder;
                signed long handle;
        };
        void *cookie;
};
```

Figure 1 shows a simplified Binder IPC transaction between two processes. Figure 2 illustrates the complete path from a userland Java application to the Binder driver. Figure 3 shows a simplified process of registering and calling an Android system service. Each service registers with the Context Manager process (ServiceManager service), which is a special Binder node with ID 0. ServiceManager is started from `init`. Android system services register with ServiceManager, and clients make requests with the ServiceManager proxy to query for system services.

`binder_ioctl()` is the entry point from userland into the kernel driver, upon which user buffers are written, read, or both. Pseudocode in Listing 1 illustrates the driver code's sequence of actions for a client making a service request. Note that `data` in steps 4c, 4d, 5b, and 5c represents a

	Dynamic Permission Blocking	Hooking Framework	Binder Hooks	Modify Kernel Binder Messages	Fine-grained Permission Revocation	Complete Mediation	Root Access Required	Custom Message Blocking	Context-informed Decisions
BinderFilter	✓	✓	✓	✓	✓	✓	✓	✓	✓
Marshmallow	✓					✓			
Xposed & XPrivacy	✓	✓			✓		✓	✓	
AppGuard	✓				✓				
TaintDroid							✓		
Boxify	✓				✓	✓			
ASM	✓	✓			✓			✓	
Man-in-the-Binder		✓	✓	✓			✓		

Table 1: BinderFilter and other Android privacy enhancement projects.

client's data being transferred to the requested service. Figure 4 illustrates this data transfer facilitated by the driver.

```
1. device_initcall(binder_init);  // called
    when kernel boots
2. binder_init()
  a. misc_register(&binder_miscdev) // register
      driver name and file operations
3. binder_ioctl() // entry point from userland
  a. wait_event_interruptable() // block caller
      until a response
  b. copy_from_user()   // copy struct
     binder_write_read from userland
  c. binder_thread_write() or
     binder_thread_read()  // depends on
     client or service request
4. binder_thread_write()  // Called by client
    making a request
  a. switch(cmd) {... // checks user command
  b. binder_transaction()
  c. copy_from_user(data) // copy data from
     userland
  *. filter_binder(data)     // our hook
  d. list_add_tail(data, target)  // add work
     to the target thread's queue
  e. wake_up_interruptable(target)  // wake up
     the sleeping service thread
5. binder_thread_read() // Called by service
    thread waiting to handle requests
  a. while (1) { if (BINDER_LOOPER_NEED_DATA)
     goto retry; }
  b. data = list_first_entry()  // get request
     data
  c. copy_to_user(data) // copy the data to
     service
```

Listing 1: Binder driver code analysis. 'data' labeled in red represents a Binder user parcel, which is moved by the Binder driver between separate process address spaces.

3. BINDERFILTER

`BinderFilter` is our hooking implementation of Binder. Compiled as a static kernel driver, our filter steals Binder messages and modifies them based on our IPC firewall policy. Being in the Binder allows us to have complete access to all IPC messages and gather context information directly from sensor hardware.

3.1 Design

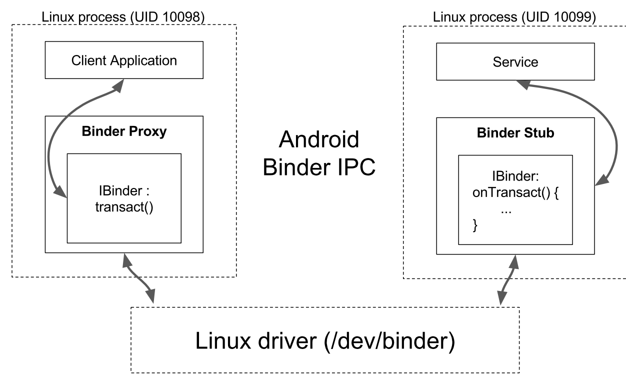

Figure 1: Simplified Binder transaction (based on [14]).

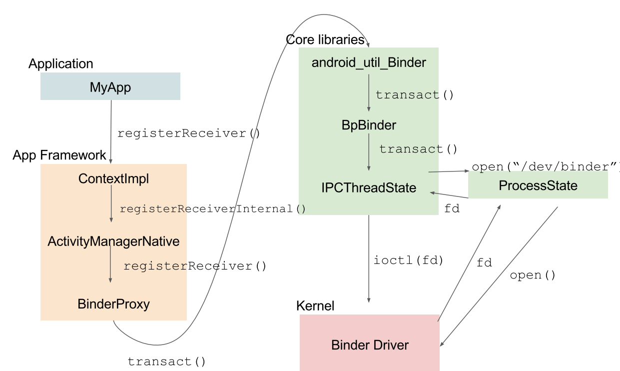

Figure 2: Binder code path

3.1.1 *Complete Mediation*

We choose to block Android permissions and "steal" IPC messages in the Binder, rather than upstream of it (as done by many of our predecessors), because this architectural choice allows us to have a good level of granularity while being able to capture all messages. Because direct Binder messages (app to app, app to service) are possible, the `ServiceManager` is not in a position of complete mediation: apps can register Binder receivers that don't go through `ServiceManager` via Service.bindService() [16].

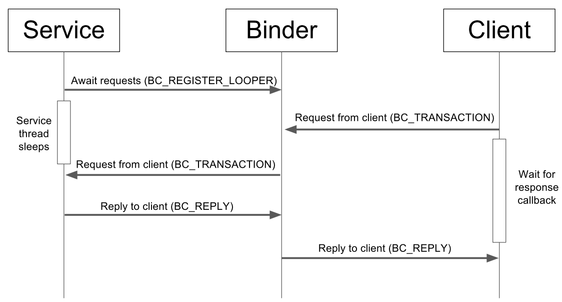

Figure 3: Binder transaction and service registration (based on [15]).

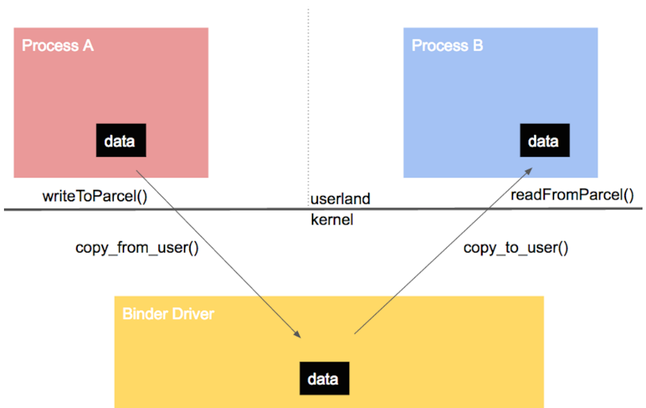

Figure 4: Transfer of userland data through the Binder. Based on Figure 5.4 from [22].

Permissions encapsulate IPC firewall policy at the level of granularity we require. For example, multiple ways to get phone location mean multiple intents to block [17]. We can block all of them with one permission! For camera messages, applications like VSCO and GoogleCamera implement their own camera wrapper, which calls Android's Camera API [18]. In this case, multiple types of intents can be encapsulated in one permission, `android.permission.CAMERA`.

3.1.2 Hook placement

We hook binder.c in one location (http://androidxref.com/kernel_3.18/xref/drivers/staging/android/binder.c#1520). At this point in a Binder call, the driver has just validated user buffer data and copied it into kernel address space, but has yet to act on it. Here we steal the buffer and modify it in the kernel if needed (Figure 5). Our hook function declaration is exported with `EXPORT_SYMBOL`.

```
#include "binder_filter.h"
extern int filter_binder_message(unsigned long,
    signed long, int, int, void*, size_t);
...
static void binder_transaction(struct
    binder_proc *proc, struct binder_thread *
    thread, struct binder_transaction_data *tr,
     int reply)
{
  struct binder_transaction *t = kzalloc(sizeof
      (*t),
                            GFP_KERNEL);
    ...
  if (copy_from_user(t->buffer->data, tr->data.
      ptr.buffer, tr->data_size)) {
    ...
    goto err_copy_data_failed;
  }
  ...
  filter_binder_message((unsigned long)(t->
      buffer->data), tr->data_size, reply, t->
      sender_euid, (void*)offp, tr->
      offsets_size);
  ...
}
```

Listing 2: Binder driver (binder.c) hook. Our additions are shown in red.

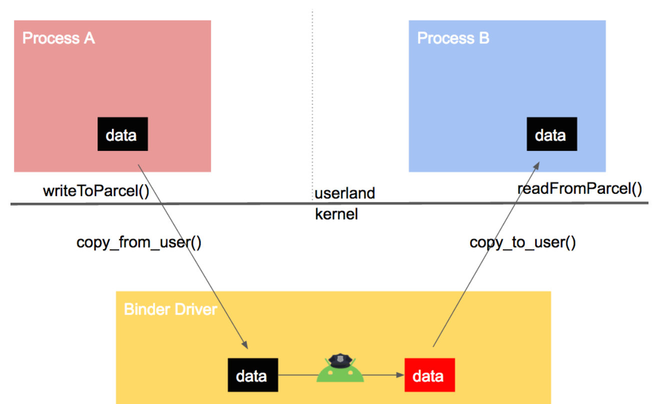

Figure 5: Binder driver hook placement

3.1.3 Grammar

We define a policy grammar for firewall rules as in Figure 6. Messages are passed in as string literals to enable support for dynamic blocking of all messages.

```
message:uid:action_code:context:(context_type:context_val:)(data:)
string  int32  int32  int32  (int32        int32/string)(string:)

android.permission.BODY_SENSORS:10008:1:0:
youtube:10061:1:0:
android.permission.CAMERA:10078:1:2:2:Dartmouth Public:
android.permission.RECORD_AUDIO:10081:3:4:1:2:meow.mp3:
```

Figure 6: BinderFilter policy grammar example. Matching fields have the same color.

3.2 Logging

The Android Binder driver (binder.c) uses three types of logging frameworks: `printk`, `TRACE_EVENT`, and `seq_printf`. We develop a log message formatter for existing Binder kernel debug message logs using Python. The code can be found at [21]. An example of the existing log message compared to its formatted version can be found in Figure 7.

Existing: [49431.544219] binder: 9916:9916 BC_TRANSACTION 683674 -> 198 - node 289403, data 8dc12180 (null) size 80-0
Pretty print: [14:33:56.084452] binder_command BC_TRANSACTION: process pid 9916 (android.picky), thread pid 9916 -> process pid 198 (/system/bin/surfaceflinger), node id 289403, transaction id 683674, data address 8dc12180, data size 80, offsets address null, offsets size 0

Figure 7: Binder log message formatting. Matching fields have the same color.

Binder's existing log output does not include buffer contents. We parse flattened Binder message buffers in BinderFilter for dynamic IPC analysis (see Figure 8). Contents are printed to the kernel debug buffer when BinderFilter's `filter_print_buffer_contents` module parameter is set.

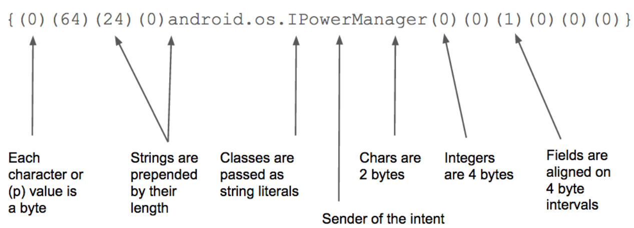

Figure 8: Binder buffer content analysis

3.3 Blocking

In this section we analyze Binder IPC message content examples much like network packets, for implementation of features specified in the introduction. Our blocking implementation looks at Binder message string literals and wipes buffer contents if the message and context match our firewall policy.

```
static void apply_filter(char* user_buf, size_t
     data_size, int euid)
{
  char* ascii_buffer =
      get_string_matching_buffer(user_buf,
      data_size);
  struct bf_filter_rule* rule = all_filters.
      filters_list_head;
  ...
  if (binder_filter_block_messages == 1) {
    while (rule != NULL) {
      if (rule->uid == euid && context_matches(
          rule)) {
        block_or_modify_messages(user_buf,
            data_size, ascii_buffer, rule->
            message);
      }
      rule = rule->next;
    }
  }
  kfree(ascii_buffer);
}

static void block_or_modify_messages(char*
    user_buf, size_t data_size, char*
    ascii_buffer, const char* message)
{
  char* message_location = strstr(ascii_buffer,
       message);
  if (message_location != NULL) {
    memset(user_buf, 0, data_size);
    }
}
```

Listing 3: BinderFilter blocking logic. UID, Context, and blocking are highlighted.

3.3.1 *Permissions*

We have currently implemented and tested blocking of the following permissions:

```
android.permission.CAMERA
android.permission.RECORD_AUDIO
android.permission.READ_CONTACTS
android.permission.WRITE_CONTACTS
android.permission.GET_ACCOUNTS
android.permission.ACCESS_FINE_LOCATION
android.permission.ACCESS_COARSE_LOCATION
android.permission.READ_EXTERNAL_STORAGE
android.permission.WRITE_EXTERNAL_STORAGE
android.permission.INTERNET
android.permission.SYSTEM_ALERT_WINDOW
android.permission.WRITE_SETTINGS
android.permission.READ_PHONE_STATE
android.permission.CALL_PHONE
android.permission.READ_CALL_LOG
android.permission.WRITE_CALL_LOG
android.permission.SEND_SMS
android.permission.RECEIVE_SMS
android.permission.READ_SMS
android.permission.RECEIVE_MMS
android.permission.RECEIVE_WAP_PUSH
android.permission.READ_CALENDAR
android.permission.WRITE_CALENDAR
android.permission.BODY_SENSORS
android.permission.ACCESS_NETWORK_STATE
android.permission.CHANGE_NETWORK_STATE
android.permission.ACCESS_WIFI_STATE
android.permission.CHANGE_WIFI_STATE
android.permission.BATTERY_STATS
android.permission.BLUETOOTH
android.permission.BLUETOOTH_ADMIN
android.permission.NFC
android.permission.FLASHLIGHT
android.permission.TRANSMIT_IR
android.permission.USE_SIP
```

The Binder message that is sent as a result of PackageManager's checkPermission() call contains the UID of the application in question and the string literal of the Android permission.

```
{(0)(64)(28)(0)android.app.IActivityManager(0)
    (0)(41)(0)android.permission.
    ACCESS_COARSE_LOCATION(0)(155)(9)(0))'(0)}
```

3.3.2 *System Permissions*

Android's `PackageManager` service checks system applications' permissions differently (see Section 2). Below is a code excerpt from the service that assigns system run-time install permissions to system apps.

```
// Only system components can circumvent
    runtime permissions when installing.
if ((installFlags & PackageManager.
    INSTALL_GRANT_RUNTIME_PERMISSIONS) != 0
&& mContext.checkCallingOrSelfPermission(
    Manifest.permission.
    INSTALL_GRANT_RUNTIME_PERMISSIONS) ==
    PackageManager.PERMISSION_DENIED) {
    throw new SecurityException("You need the "
```

```
            + "android.permission.
                INSTALL_GRANT_RUNTIME_PERMISSIONS
                 permission "
            + "to use the PackageManager.
                INSTALL_GRANT_RUNTIME_PERMISSIONS
                 flag");
}
```

Listing 4: PackageManagerService.installPackageAsUser(). System package permissions check is highlighted.

To properly block system permissions, we must look at the specific apps that use them. An analysis of Google Play Store finds that before installation of packages, the `com.android.vending.INTENT_PACKAGE_INSTALL_COMMIT` intent is sent. Here we can block that intent in lieu of the permission.


```
{(0)(64)(28)(0)android.app.IActivityManager(0)(0)(1)(0)(19)(
0)com.android.vending(0)(133)*bs(127)(1)(0)(0)(0)(0)(0)(255)
(255)(255)(255)k(157)+m(1)(0)(1)(0)(1)(0)E(0)com.android.ven
ding.INTENT_PACKAGE_INSTALL_COMMIT.com.groupme.android(0)(0)
(0)(255)(255)(255)(255)(0)(0)(255)(255)(255)(255)(255)(255)(
255)(255)(0)(0)(0)(0)(0)(0)(0)(0)(254)(255)(255)(255)(255)(2
55)(255)(255)(255)(255)(255)(255)(0)(0)H(0)(0)(0)(0)}
```


Figure 9: Message analysis for installation activity

3.4 Context

Context is obtained directly from sensor data in the BinderFilter driver. This defends against spoofed data. In the event that context information cannot be obtained, default fallback policy rules are followed. Figure 10 shows parsing wifi SSID from its flattened Binder buffer.


```
{(0)@(30)(0)android.app.IApplicationThread(0)(0)(133)h(127)
(07)(247)(07)(07)(07)$(07)android.net.conn.CONNECTIVITY_CHANGE
(07)(07)(07)(07)(255)(255)(16)(0)(255)(255)(255)(255)(05)(05)
(05)(05)(05)(05)(05)(05)(254)(255)x(05)BD(45)(05)(11)(0)
networkInfo(0)(4)(0)(23)(0)android.net.NetworkInfo(0)(1)(0)(0)
(0)(4)(0)WIFI(0)(0)(0)(0)(0)(0)(9)(0)CONNECTED(0)(9)(0)
CONNECTED (0)(0)(0)(1)(0)(0)(0)(255)(255)(18)(0)"SecureNet"(0)
(0)(11)(0)networkType(0)(1)(0)(1)(0)(13)(0)inetCond}
```


Figure 10: Message analysis for Wi-Fi SSID

3.5 Message Modification

By *modification* of Binder messages we mean replacing message content with other content before the (modified) message reaches its target. For example, a user may want to send fake image or audio recording data to an application instead of blocking requests and risking a crash, should the application fail to deal gracefully with the failure.

This capability powerfully complements blocking of messages and has been implemented by the best-of-breed host-based network firewalls such as *Netfilter*, which includes the *IPQUEUE* and newer *NFQUEUE* mechanisms to just this effect. These mechanisms also enable low-level security research, as we hope ours will as well.

We implement copying file contents with `sys_read` and `sys_write`.

```
static void copy_file_to_file(char*
    filename_src, char* filename_dst)
{
  ...
  set_fs(KERNEL_DS);
  fd_read = sys_open(filename_src, O_RDONLY, 0)
      ;
  fd_write = sys_open(filename_dst, O_WRONLY|
      O_CREAT|O_TRUNC, 0644);
  ...
  while (1) {
    read_len = sys_read(fd_read, read_buf,
        buf_size-1);
    if (read_len <= 0) {
      break;
    }
    sys_write(fd_write, read_buf, read_len);
    write_file = fget(fd_write);
    ...
    vfs_write(write_file, read_buf, read_len, &
        pos);
        fput(write_file);
  }
  ...
}
```

Listing 5: Code snippet of buffered file copying in the kernel


```
{(4)H(28)(0)android.app.IActivityManager(0)(0)(133)*bs(127)(
1)(0)P(196)(180)(174)(224)(145)(181)(172)(19)(0)com.facebook
.katana(0)"(0)android.media.action.IMAGE_CAPTURE(0)(0)(0)(0)
(255)(255)(255)(255)(3)(0)(255)(255)(255)(255)(255)(255)(255
)(255)(0)(0)(0)(0)(0)(0)(1)(0)(1)(0)(0)(0)(0)(0)(1)(0)(13)(0
)text/uri-list(0)(0)(0)(1)(0)(1)(0)(255)(255)(255)(255)(255)
(255)(255)(255)(0)(0)(1)(0)(3)(0)(4)(0)file(0)(0)(0)(0)(0)(0
)(0)(0)(0)(0)(0)(0)(2)(0)(62)(0)/storage/emulated/0/Pictures
/Facebook/FB_IMG_1464314001208.jpg}
```


Figure 11: Message analysis for image capture intent

3.6 Deployment

Android versions 4.3 and above disable loadable kernel modules by default. To hook Binder, which is a statically compiled kernel driver, we must recompile the kernel with our hooking code in it. We can then flash the new kernel image onto an Android device. This step preserves user information, apps, and state, and requires an unlocked bootloader and root access.

4. PICKY

Picky implements a user interface for setting BinderFilter policy. It allows users to dynamically set policy in an accessible and usable way. Supported features include:

- Requiring a lockscreen to open
- Import and Export policy file
- Persistent policy across application sessions and reboot
- Per-application blocking of messages
- User-defined custom messages
- Contextual policy rules
- Modification of messages

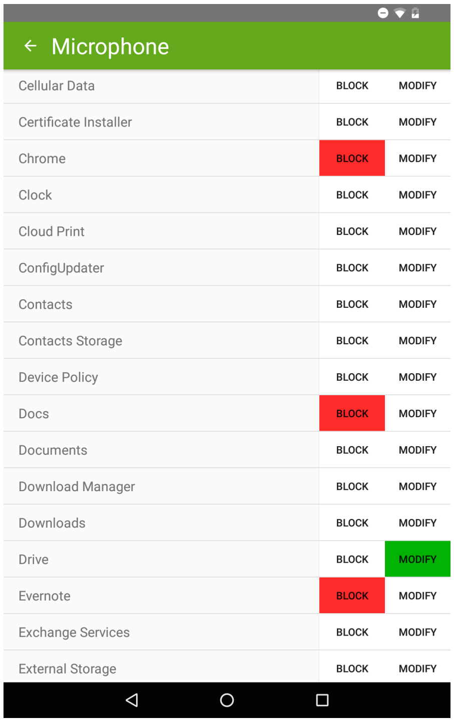

Figure 12: Per-application blocking

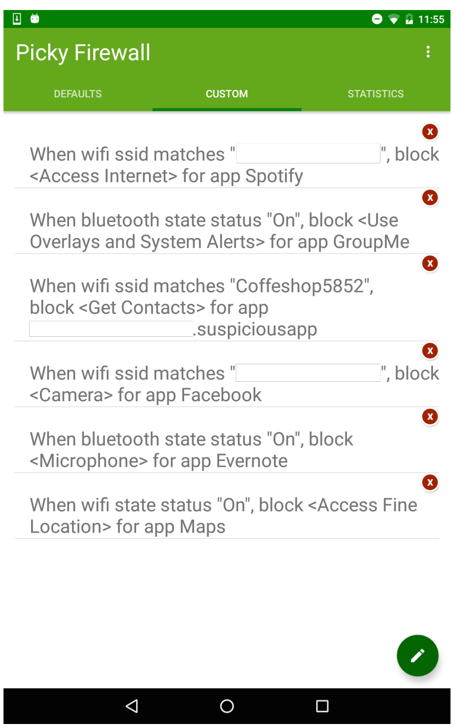

Figure 13: User-defined context rules

4.1 Kernel Interface

Android's NDK (Native Development Kit) allows Java apps to call Native C++ code through the JNI (Java Native Interface) framework [20]. Java functions with the `native` keyword are implemented in the JNI layer. We use this layer to call `sys_open`, `sys_read`, and `sys_write` to read and write userland policy to and from the BinderFilter kernel driver. File and SELinux permissions are dynamically set for BinderFilter drivers to allow access.

```
Picky.java:
  public static native String nativeReadPolicy
      ();
  public static native String
      nativeWriteFilterLine(int action, int uid
      , String message, String data);

picky-jni.c:
  JNIEXPORT jstring JNICALL
  Java_Picky_Policy_nativeReadPolicy(JNIEnv *
      env, jclass type) {
    char returnValue[4096]
    ...
    int fd = open("/dev/binderfilter", O_RDWR);
    int len = read(fd, returnValue, sizeRead);
    ...
    return (*env)->NewStringUTF(env,
        returnValue);
  }

  JNIEXPORT jstring JNICALL
  Java_Picky_Policy_nativeWriteFilterLine(
      JNIEnv *env, jclass type, jint action,
      jint uid, jstring message_, jstring data_
      ) {
    struct bf_user_filter user_filter;
    user_filter.action = (int) action;
    user_filter.uid = (int) uid;
    user_filter.message = (char*) message;
    user_filter.data = (char*) data;
    user_filter.context = 0;
    int fd = open("/dev/binderfilter", O_RDWR);
    int write_len = write(fd, &user_filter,
        sizeof(user_filter));
    ...
    return (*env)->NewStringUTF(env,
        write_len_str);
  }

binder_filter.c:
  static ssize_t bf_write(struct file *file,
      const char __user *buf, size_t len,
      loff_t *ppos)
  {
    struct bf_user_filter* user_filter =
        init_bf_user_filter();
    ...
    if (copy_from_user(user_filter, buf, sizeof
        (struct bf_user_filter))) {
      return 0;
    }
    ...
    add_or_remove_filter(user_filter);
    write_persistent_policy();
    ...
  }

  static ssize_t bf_read(struct file * file,
      char * buf, size_t count, loff_t *ppos)
  {
    int len;
    char* ret_str = get_policy_string();
```

```
    len = strlen(ret_str);
    ...
    if (copy_to_user(buf, ret_str, len)) {
      return -EINVAL;
    }
    ...
    return len;
}
```

Listing 6: Android JNI layer kernel interaction (highlighted).

5. EVALUATION

Here we discuss the effectiveness of our firewall against various classes of attacks previously discussed in both academic and industry literature.

Information stealing and overzealous applications. We tested our firewall with various applications from the Play Store such as Facebook, Evernote, Keep, Maps, etc., as well as an application we developed as a sanity check on individual permissions. We note that the adoption of Android 6.0 dynamic permission checking by application developers has not been widespread, and, whereas some apps like Facebook and Evernote will restart gracefully on a permission denial, other applications will stop due to a lack of dynamic permission checking. Both stock applications and malicious applications such as Android.Enesoluty [31] and Android.Loozfon [32] are subjected to our message filter.

Malicious apps with root privileges such as mempodroid [33]. Root applications are contained by Linux processes with the root or system UID. Because we intercept all messages for all UIDs, apps started with the system UID are blocked by our firewall regardless of root privileges. We note that if malicious applications are able to overwrite underlying SELinux policies, they could disable our firewall; however, preventing such priviliege escalation in the filesystem is out of the scope of this project.

Collusion attacks such as [34, 35]. We can block applications that combine their separate permissions in order to achieve functionality beyond what a user intends. We detect an application's process state as started or stopped based on the `android.intent.category.LAUNCHER` and `android.intent.action.PACKAGE_RESTARTED` intents. This application running context can then inform policy to prevent colluding applications.

UI-based attacks. These attacks trick users into providing input into a window controlled by the attacker, either by overlaying malicious elements on top of trusted applications [36, 37, 38, 39], or by preempting trusted applications with a phishing window [36, 38, 40, 41, 42]. Android 6.0 requires users to explicitily allow overlay permissions in response to many abuses of the overlay architecture. For previous versions of Android, our IPC firewall blocks overlay access with the `android.permission.SYSTEM_ALERT_WINDOW` permission.

We can prevent *activity hijacking* attacks that preempt trusted application screens with an identical phishing screen. Our firewall blocks WindowManager requests for an application during that application's window session. This is analogous to the Overlay Mutex in [36], which only allows one application to control the display window at once. We define a window session start as an application intent to IWindowSession, which contains the package name for the starting application. The window session end is defined as a user touch input event for either the back, home, or menu button. These events are captured in the Binder from `android.hardware.input.IInputManager`.

6. RELATED WORK

Research around Android security systems has have focused on data privacy in terms of permission revocation [8, 24, 25, 26, 27, 28, 29], taint analysis [9, 23], and sandboxing [10, 24, 25]. Taint analysis was first introduced in Enck et al.'s influential `TaintDroid`, which monitors user data by tagging (tainting) it dynamically in Android's Dalvik VM [9]. LeakMiner [23] uses static taint analysis for applications on the Play Store to detect possible information leakage. Permission revocation has been used to prevent information leakage via an inline reference monitor in Backes et al. [8]. Aurasium achieves fine-grained, dynamic permission checking by repackaging application packages with sandboxing code, which is then enforced by hooks in the Dalvik VM [24]. Backes et al. (2015) deviates from UID-permission based security architectures with an application sandboxing architecture based on app virtualization [10]. FlaskDroid [25] enforces fine-grained, dynamic permission revocation in the kernel by dynamically setting SELinux policies for userland applications based on a trusted user space agent. We take their approach of a trusted userland policy application enforced by the kernel.

Although our Binder IPC hooking approach requires modification to underlying kernel code, it makes no assumptions about Android layers that are more likely to change over time, such as the Dalvik VM. Specifically, security add-ons that rely on Dalvik hooks may be broken in by Android's successor to Dalvik, ART, which introduced ahead-of-time compilation in Android 5.0.

Previous architectures that have hooked Binder have shown promising results: we build on projects like DeepDroid [29] to systematically intercept and modify all Binder messages without relying on Dalvik VM or Android middleware code injection. Nitay Artenstein and Idan Revivo demonstrated Binder message parsing and data exfiltration but did not go as far as systematically hooking and modifying message data [13].

We adopt DeepDroid and CRePE's automatic context detection for policy enforcement [29, 30] and apply it to the new Android security landscape of SELinux, ART, and dynamic permission revocation in Marshmallow. Security policy must be informed by the environment that a system encounters. To summarize, our main contributions to this previous work include a context-aware policy system that bases filtering decisions on the system's state, and a hooking framework in Android's IPC system that supports blocking, stealing, and modification of all IPC messages.

7. CONCLUSIONS

We have introduced a new hooking framework for Binder. Our use of Binder driver hooks in BinderFilter and Picky allows users to control their privacy settings per application, context, and message. In addition, users can specify custom messages to block and modify saved content data. More work remains to be done in adding contexts and message modifications. Because dynamic permission checking was only recently introduced in Android 6.0, many apps still do not handle permission revocation, leading to crashes. Modification of message content (as opposed to simply blocking messages) could prevent unwanted crashes of applications while also ensuring user data privacy.

8. REFERENCES

[1] David Wu. BinderFilter. 2016. https://github.com/dxwu/AndroidBinder

[2] David Wu. Picky. 2016. https://github.com/dxwu/Picky

[3] The Verge. Android is now used by 1.4 billion people. 2015. http://www.theverge.com/2015/9/29/9409071/google-android-stats-users-downloads-sales

[4] The Boston Globe. Harvard, Northeastern's privacy tools flag apps that leak personal data. November 16, 2015.

[5] The Wall Street Journal. Facebook Messenger Privacy Fears? Here's What to Know. August 8, 2014.

[6] Google. Android - Marshmallow. 2016. https://www.android.com/versions/marshmallow-6-0/

[7] Google. Android - Dashboards. 2016. https://developer.android.com/about/dashboards/index.html

[8] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard–Real-time policy enforcement for third-party applications. Technical Report. 2012. https://www.infsec.cs.uni-saarland.de/projects/appguard/android_irm.pdf

[9] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaneil, Anmod N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. 2014. Commun. ACM. https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Enck.pdf

[10] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. 2015. USENIX 2015. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-backes.pdf

[11] Stephan Heuser, Adwait Nadkarni, William Enck and Ahmad-Reza Sadeghi. ASM: A Programmable Interface for Extending Android Security. 2014. USENIX 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-heuser.pdf

[12] rovo89. Xposed. 2016. http://repo.xposed.info/module/de.robv.android.xposed.installer

[13] Nitay Artenstein and Idan Revivo. Man in the Binder: He Who Controls IPC, Controls the Droid. 2014. Blackhat 2014.

[14] Nikolay Elenkov. Android Security Internals. No Starch Press, 2015

[15] Aleksandar Gargenta. Deep Dive into Android IPC/Binder Framework. 2013. Android Builders Summit 2013. http://events.linuxfoundation.org/images/stories/slides/abs2013_gargentas.pdf

[16] Google. Android - Creating a Bound Service. 2016. https://developer.android.com/guide/components/bound-services.html#Creating

[17] Google. Android - Location Strategies. 2016. https://developer.android.com/guide/topics/location/strategies.html

[18] Google. Android - Camera API. 2016. https://developer.android.com/reference/android/hardware/Camera.html

[19] AndroidXRef. PackageManagerService source code. 2016. http://androidxref.com/6.0.1_r10/xref/frameworks/base/services/core/java/com/android/server/pm/PackageManagerService.java#9557

[20] Android Studio Project. Android NDK Preview. http://tools.android.com/tech-docs/android-ndk-preview

[21] David Wu. PrettyPrintBinder. https://github.com/dxwu/AndroidBinder/blob/master/PrettyPrintBinder.py

[22] Thorsten Schreiber. Android Binder. 2011. Ruhn-Universitat Bochum. http:www.nds.rub.du/media/attachments/files/2012/03/binder.pdf

[23] Z. Yang and M. Yang. Leakminer: Detect information leakage on android with static taint analysis. WCSE, 2012

[24] R. Xu, H. Saidi, R. Anderson. Aurasium: practical policy enforcement for Android applications. USENIX Security, 2012

[25] Sven Bugiel, Stephan Heuser and Ahmad-Reza Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. 2013. USENIX 2013.

[26] Nauman, M., Khan, S., and Zhang, X. Apex: Extending nandroid permission model and enforcement with user-defined runtime constraints. In Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS'10) (2010), ACM.

[27] Ongtang, M., McLaughlin, S. E., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In Proc. 25th Annual Computer Security Applications Conference (ACSAC'09) (2009), ACM.

[28] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. Taming information-stealing smartphone applications (on Android). In Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST'11) (2011), Springer.

[29] Wangy, X., Sun, K., and Jing, Y. W. J. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In Proc. 22nd Annual Network and Distributed System Security Symposium (NDSS'15) (2015), The Internet Society.

[30] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. CRePE: Context-Related Policy Enforcement for Android. In Proc. 13th International Conference on Information Security, ISC'10, (2011), Springer. pp. 331–345

[31] Asuka Yamamoto. Symantec. Android.Enesoluty. http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99

[32] Santiago Cortes. Symantec. Android.Loozfon. http://www.symantec.com/security_response/writeup.jsp?docid=2012-082005-5451-99

[33] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to

Android. In Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13) (2013), The Internet Society.
[34] A. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In USENIX Security (2011), USENIX
[35] Claudio Marforio, Aurelien Francillon, Srdjan Capkun. Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems. Technical Report 724, ETH Zurich, 2011.
[36] Earlence Fernandes, Qi Alfred Chen, Justin Paupore, Georg Essl, J. Alex Halderman, Z. Morley Mao, Atul Prakash. Android UI Deception Revisited: Attacks and Defenses. In 20th International Conference on Financial Cryptography and Data Security (FC'16), Barbados, February 2016.
[37] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, Giovanni Vigna. What the App is That? Deception and Countermeasures in the Android User Interface. In: Proceedings of the IEEE Symposium on Security and Privacy (SP). San Jose, CA (May 2015)
[38] Qi Alfred Chen, Zhiyun Qian, Z. Morley Mao. Peeking into Your App without Actually Seeing It. UI State Inference and Novel Android Attacks. In: Proceedings of the 23rd USENIX Security Symposium (2014)
[39] Earlence Fernandes, Qi Alfred Chen, Georg Essl, J. Alex Halderman, Z. Morley Mao, Atul Prakash. Trusted Visual I/O Paths for Android. Tech. Rep. Technical Report CSETR-586-14, CSE Department, University of Michigan, Ann Arbor (2014)
[40] Kaspersky. Svpeng android malware targets banking apps. 2014. http://www.kaspersky.com/about/news/virus/2014/Kaspersky-Lab-detects-mobile-TrojanSvpeng-Financial-malware-with-ransomware-capabilities-now-targeting-US-users
[41] Activity hijacking pattern for Android. http://capec.mitre.org/data/definitions/501.html
[42] Erika Chin, Adrienne Porter Felt, Kate Greenwood, David Wagner. Analyzing Inter-application Communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services. pp. 239–252. MobiSys'11, ACM, 2011.

APPENDIX

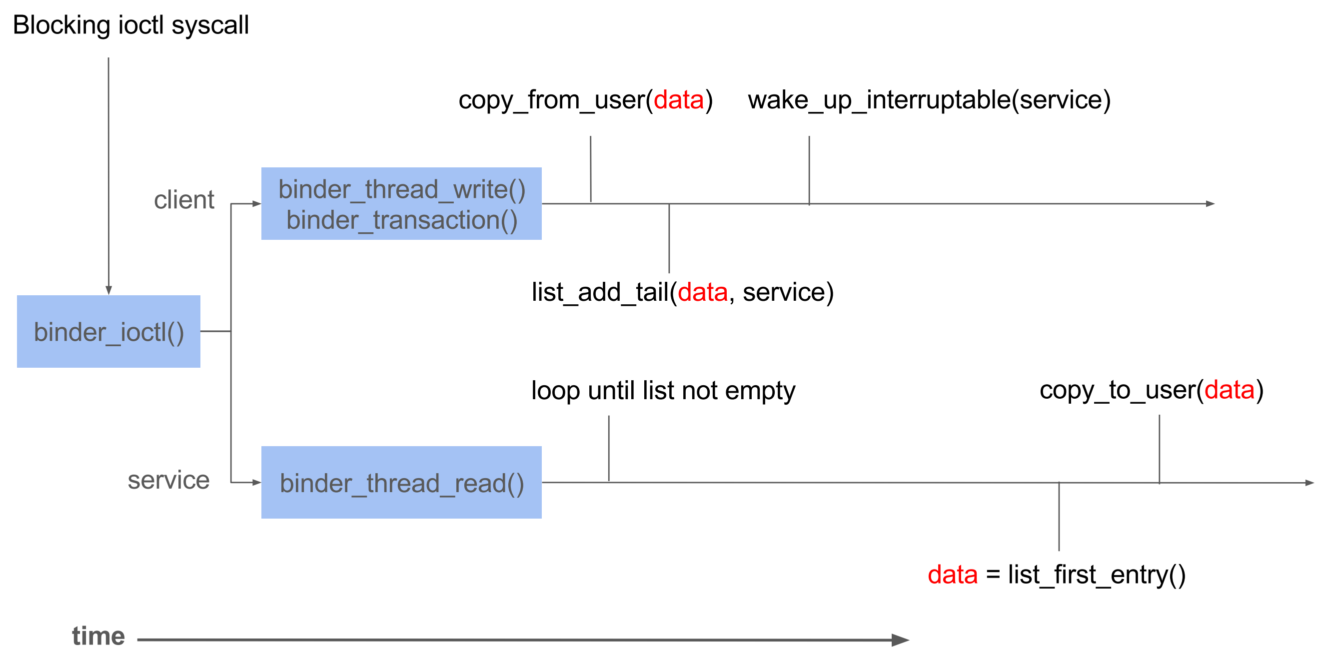

Figure 14: Binder driver code analysis timeline. Visualization of Listing 1.

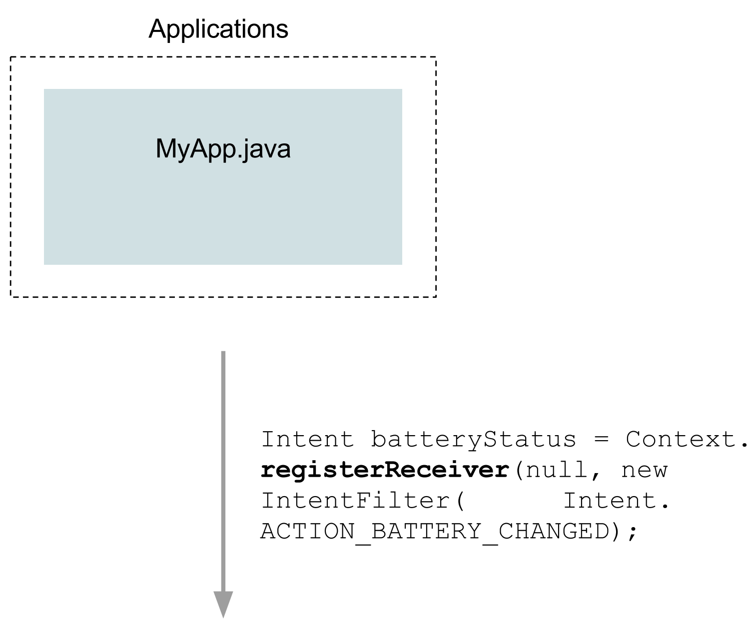

Figure 15: Binder code path (application layer). Expansion of Figure 2.

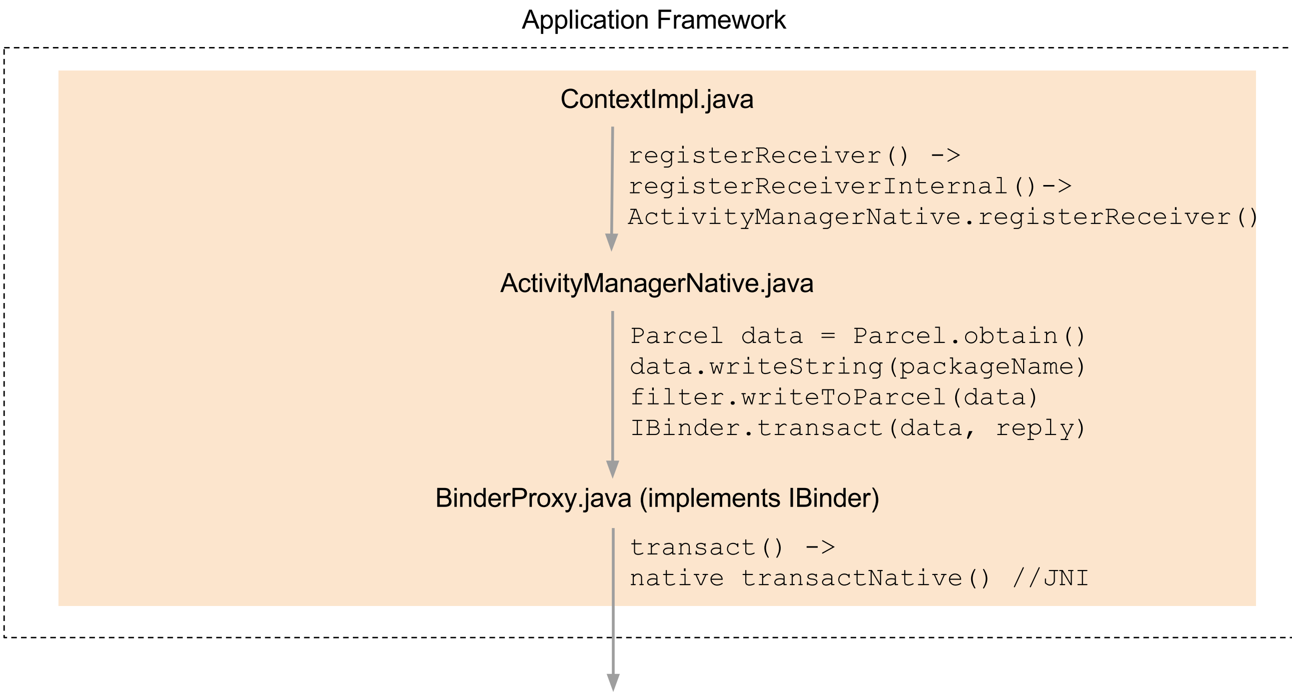

Figure 16: Binder code path (application framework layer). Expansion of Figure 2..

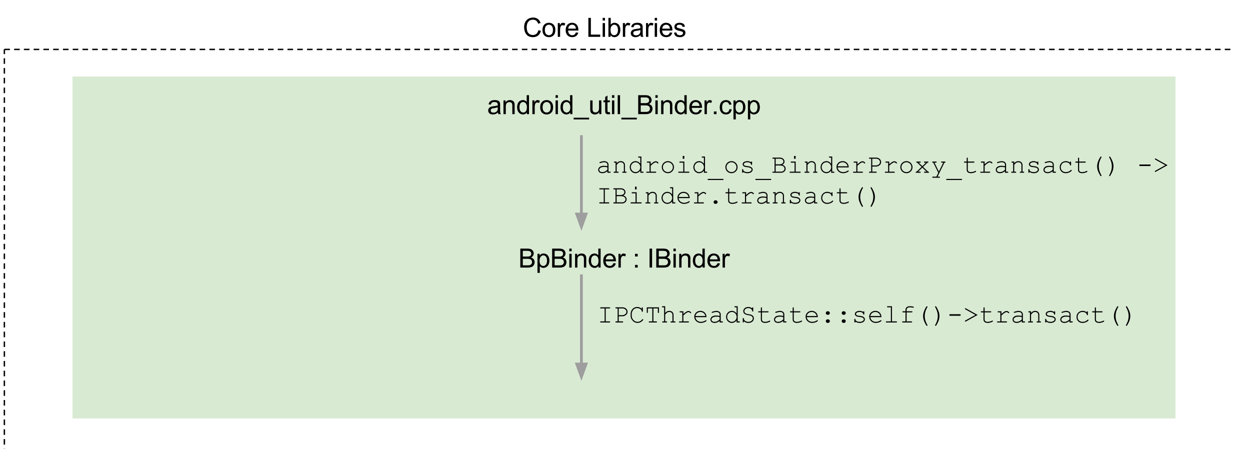

Figure 17: Binder code path (core libraries layer). Expansion of Figure 2.

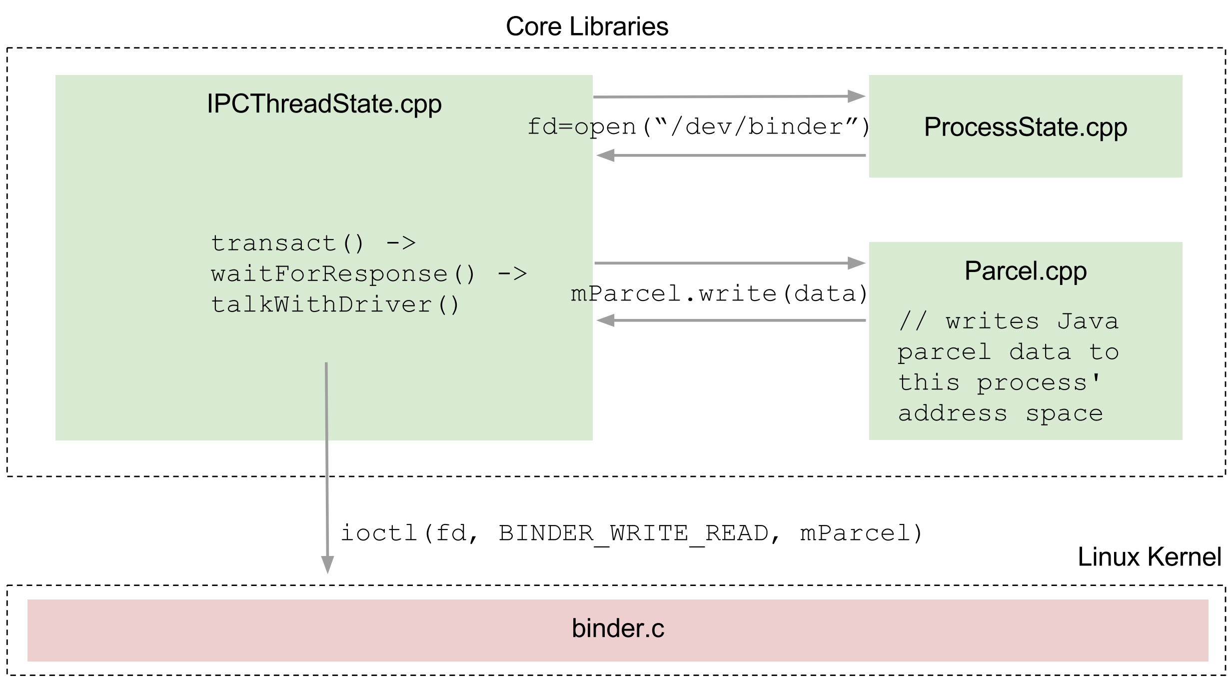

Figure 18: Binder code path (core libraries continued and linux kernel layer). Expansion of Figure 2.