

# System-level attacks against android by exploiting asynchronous programming

Ting Chen<sup>1</sup>  · Xiaoqi Li<sup>2</sup> · Xiapu Luo<sup>2</sup> ·  
Xiaosong Zhang<sup>1</sup>

© Springer Science+Business Media New York 2017

**Abstract** To avoid unresponsiveness, Android developers utilize asynchronous programming to schedule long-running tasks in the background. In this work, we conduct a systematic study on IntentService, one of the async constructs provided by Android using static program analysis, and find that in Android 6, 974 intents can be sent by third-party applications without protection. Based on this observation, we develop a tool, ATUIN, to demonstrate the feasibility of attacking a CPU automatically by exploiting the intents that can be handled by an Android system. Furthermore, by investigating the unprotected intents, we discover tens of critical vulnerabilities that have not been reported before, including Wi-Fi DoS, telephone signal blocking, SIM card removal, homescreen hiding, and NFC state cheating. Our study sheds light on research into protecting asynchronous programming from being exploited by hackers.

**Keywords** Asynchronous programming · Android · IntentService · System-level attacks · Wi-Fi DoS · Telephone signal block · SIM card removal · Homescreen hiding · NFC state cheating

---

✉ Ting Chen  
brokendragon@uestc.edu.cn

Xiaoqi Li  
csxqli@comp.polyu.edu.hk

Xiapu Luo  
csxluo@comp.polyu.edu.hk

Xiaosong Zhang  
johnsonzxs@uestc.edu.cn

<sup>1</sup> Cybersecurity Research Center, University of Electronic Science and Technology of China, Chengdu, China

<sup>2</sup> Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong

## 1 Introduction

Responsiveness is critical for smartphones. However, smartphones are likely to be unresponsive because of their limited computing resources and frequent network operations. Previous researches show that many Android applications suffer from poor responsiveness and one of the primary reasons is that applications run too much workload in the UI event thread (Liu et al. 2014; Yang et al. 2013). The primary way to avoid unresponsiveness is to resort to concurrency that puts long-running tasks into background threads and runs the main thread and the background threads asynchronously.

To make asynchronous programming easier, Android provides three major async constructs: `AsyncTask`, `IntentService`, and `AsyncTaskLoader`. `AsyncTask` is designed for short-running tasks while the other two are good choices for long-running tasks. Although `AsyncTask` is the most widely used construct, it may result in memory leaks, lost results, and wasted energy if improperly used (Lin et al. 2014, 2015). The other two constructs do not suffer from the same problems encountered by `AsyncTask` because they do not hold a reference to GUI (Lin et al. 2015). `AsyncTaskLoader` is introduced after Android 3.0, and it only supports two GUI components: activity and fragment. Hence, Lin et al. developed ASYNCNDROID (Lin et al. 2015) to refactor `AsyncTask`-related code into using `IntentService`, a more general and safer async construct.

We conduct a systematic study on `IntentService` to check whether the async construct is used properly and whether hackers can take advantage of unprotected intents to launch attacks. We find that in Android 6, 974 intents are not well protected and hence can be sent by third-party applications. Based on this observation, we develop a tool, ATUIN (short for ATtacks by exploiting Unprotected INtents), to demonstrate the feasibility of attacking CPU automatically by periodically sending unprotected intents that can be processed by Android system. Furthermore, by inspecting unprotected intents, we discover tens of critical vulnerabilities that have not been reported before, such as Wi-Fi DoS, telephone signal block, SIM card removal, homescreen hiding, and NFC state cheating.

Overall, our study has three major contributions:

- We conduct the *first* systematic study on `IntentService`, and discover nearly 1000 unprotected intents in Android 6, which could be exploited to launch Denial-of-Service attacks on the system.
- We develop ATUIN to demonstrate the feasibility of attacking CPU automatically by periodically sending unprotected intents that can be handled by Android system.
- We further discover tens of critical unreported system vulnerabilities that can disable some key functionalities of smartphones (e.g., Wi-Fi, telephone, launch activity).

The rest of this paper is organized as follows. Section 2 gives a motivating example. Section 3 briefly introduce background knowledge. Section 4 presents the approach and results of the systematic study. Section 5 details the implementation of ATUIN and its experimental results. Section 6 elaborates on the discovered vulnerabilities. We discuss the threats to validity in Section 7. Related studies are briefly discussed in Section 8. This paper concludes in Section 9.

## 2 Motivating example

This section shows a real vulnerability in Android 6 which involves an unprotected intent `ACTION_STEP_IDLE_STATE`. By exploiting the intent, any third-party applications

without requiring any permissions can force a smartphone to leave IDLE state immediately after it enters IDLE state. Therefore, hackers can deplete the battery power of Android phones quickly.

To save power, Android 6 introduces a new feature, so-called Doze mode, which is able to reduce power consumption aggressively by forbidding or deferring critical tasks when the smartphone is in IDLE state. In implementation, Android 6 defines seven states, in which only IDLE state can save power. Figure 1 shows a part of the code implementing state transitions.

More specifically, the core source file of Doze mode is `/services/core/java/com/android/server/DeviceIdleController.java`, which defines a pending intent `ACTION_STEP_IDLE_STATE`. When a pre-established time slice expires, the `AlarmManager` sends the intent to trigger state transition. We can see that `DeviceIdleController.java` registers a broadcast receiver (Line 240), and if it receives the `ACTION_STEP_IDLE_STATE` intent (Line 253), the function `stepIdleStateLocked` (Line 255) will be invoked. The code (Line 1266 to 1277) demonstrates that Android system transfers from `INACTIVE` state (Line 1266) to `IDLE_PENDING` state (Line 1274) once the `stepIdleStateLocked` is invoked.

However, the implementation of Doze mode is vulnerable since the critical intent `ACTION_STEP_IDLE_STATE` is unprotected, indicating that any third-party applications can send the intent without requiring any permissions. Therefore, an attacker can deplete the battery quickly by tricking innocents to install the malware which detects current state and then sends the specific intent if Android is in IDLE. We have to mind that tricking users to install the malware would not be difficult since the malware does not need any permissions. A 3-h testing shows that an LG Nexus 5X under attack consumes 6X more power than the normal situation. A detailed description of the attack can refer to our previous work (Chen et al. 2016).

### 3 Background

This section introduces the background knowledge closely related to this study. First, we present the overall architecture of the Android system. Then, we focus on the four

```

240  private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
241      @Override public void onReceive(Context context, Intent intent) {
242          ...
243          } else if (ACTION_STEP_IDLE_STATE.equals(intent.getAction())) {
244              synchronized (DeviceIdleController.this) {
245                  stepIdleStateLocked();
246              }
247          ...
248      }
249
250      void stepIdleStateLocked() {
251          ...
252          switch (mState) {
253              case STATE_INACTIVE:
254                  // We have now been inactive long enough, it is time to start looking
255                  // for significant motion and sleep some more while doing so.
256                  startMonitoringSignificantMotion();
257                  scheduleAlarmLocked(mConstants.IDLE_AFTER_INACTIVE_TIMEOUT, false);
258                  // Reset the upcoming idle delays.
259                  mNextIdlePendingDelay = mConstants.IDLE_PENDING_TIMEOUT;
260                  mNextIdleDelay = mConstants.IDLE_TIMEOUT;
261                  mState = STATE_IDLE_PENDING;
262                  if (DEBUG) Slog.d(TAG, "Moved from STATE_INACTIVE           STATE_IDLE_PENDING.");
263                  EventLogTags.writeDeviceIdle(mState, "step");
264                  break;
265              case STATE_IDLE_PENDING:
266                  mState = STATE_SENSING;
267                  if (DEBUG) Slog.d(TAG, "Moved from STATE_IDLE_PENDING to STATE_SENSING.");
268                  EventLogTags.writeDeviceIdle(mState, "step");
269                  scheduleSensingAlarmLocked(mConstants.SENSING_TIMEOUT);
270          }
271      }
272
273      synchronized (DeviceIdleController.this) {
274          ...
275          if (mState == STATE_IDLE_PENDING) {
276              if (mNextIdlePendingDelay < System.currentTimeMillis())
277                  return;
278          }
279          ...
280      }
281
282      synchronized (DeviceIdleController.this) {
283          ...
284          if (mState == STATE_SENSING) {
285              if (mNextIdleDelay < System.currentTimeMillis())
286                  return;
287          }
288          ...
289      }
290
291      synchronized (DeviceIdleController.this) {
292          ...
293          if (mState == STATE_SENSING) {
294              if (mNextIdlePendingDelay < System.currentTimeMillis())
295                  return;
296          }
297          ...
298      }
299
300      synchronized (DeviceIdleController.this) {
301          ...
302          if (mState == STATE_SENSING) {
303              if (mNextIdlePendingDelay < System.currentTimeMillis())
304                  return;
305          }
306          ...
307      }
308
309      synchronized (DeviceIdleController.this) {
310          ...
311          if (mState == STATE_SENSING) {
312              if (mNextIdlePendingDelay < System.currentTimeMillis())
313                  return;
314          }
315          ...
316      }
317
318      synchronized (DeviceIdleController.this) {
319          ...
320          if (mState == STATE_SENSING) {
321              if (mNextIdlePendingDelay < System.currentTimeMillis())
322                  return;
323          }
324          ...
325      }
326
327      synchronized (DeviceIdleController.this) {
328          ...
329          if (mState == STATE_SENSING) {
330              if (mNextIdlePendingDelay < System.currentTimeMillis())
331                  return;
332          }
333          ...
334      }
335
336      synchronized (DeviceIdleController.this) {
337          ...
338          if (mState == STATE_SENSING) {
339              if (mNextIdlePendingDelay < System.currentTimeMillis())
340                  return;
341          }
342          ...
343      }
344
345      synchronized (DeviceIdleController.this) {
346          ...
347          if (mState == STATE_SENSING) {
348              if (mNextIdlePendingDelay < System.currentTimeMillis())
349                  return;
350          }
351          ...
352      }
353
354      synchronized (DeviceIdleController.this) {
355          ...
356          if (mState == STATE_SENSING) {
357              if (mNextIdlePendingDelay < System.currentTimeMillis())
358                  return;
359          }
360          ...
361      }
362
363      synchronized (DeviceIdleController.this) {
364          ...
365          if (mState == STATE_SENSING) {
366              if (mNextIdlePendingDelay < System.currentTimeMillis())
367                  return;
368          }
369          ...
370      }
371
372      synchronized (DeviceIdleController.this) {
373          ...
374          if (mState == STATE_SENSING) {
375              if (mNextIdlePendingDelay < System.currentTimeMillis())
376                  return;
377          }
378          ...
379      }
380
381      synchronized (DeviceIdleController.this) {
382          ...
383          if (mState == STATE_SENSING) {
384              if (mNextIdlePendingDelay < System.currentTimeMillis())
385                  return;
386          }
387          ...
388      }
389
390      synchronized (DeviceIdleController.this) {
391          ...
392          if (mState == STATE_SENSING) {
393              if (mNextIdlePendingDelay < System.currentTimeMillis())
394                  return;
395          }
396          ...
397      }
398
399      synchronized (DeviceIdleController.this) {
400          ...
401          if (mState == STATE_SENSING) {
402              if (mNextIdlePendingDelay < System.currentTimeMillis())
403                  return;
404          }
405          ...
406      }
407
408      synchronized (DeviceIdleController.this) {
409          ...
410          if (mState == STATE_SENSING) {
411              if (mNextIdlePendingDelay < System.currentTimeMillis())
412                  return;
413          }
414          ...
415      }
416
417      synchronized (DeviceIdleController.this) {
418          ...
419          if (mState == STATE_SENSING) {
420              if (mNextIdlePendingDelay < System.currentTimeMillis())
421                  return;
422          }
423          ...
424      }
425
426      synchronized (DeviceIdleController.this) {
427          ...
428          if (mState == STATE_SENSING) {
429              if (mNextIdlePendingDelay < System.currentTimeMillis())
430                  return;
431          }
432          ...
433      }
434
435      synchronized (DeviceIdleController.this) {
436          ...
437          if (mState == STATE_SENSING) {
438              if (mNextIdlePendingDelay < System.currentTimeMillis())
439                  return;
440          }
441          ...
442      }
443
444      synchronized (DeviceIdleController.this) {
445          ...
446          if (mState == STATE_SENSING) {
447              if (mNextIdlePendingDelay < System.currentTimeMillis())
448                  return;
449          }
450          ...
451      }
452
453      synchronized (DeviceIdleController.this) {
454          ...
455          if (mState == STATE_SENSING) {
456              if (mNextIdlePendingDelay < System.currentTimeMillis())
457                  return;
458          }
459          ...
460      }
461
462      synchronized (DeviceIdleController.this) {
463          ...
464          if (mState == STATE_SENSING) {
465              if (mNextIdlePendingDelay < System.currentTimeMillis())
466                  return;
467          }
468          ...
469      }
470
471      synchronized (DeviceIdleController.this) {
472          ...
473          if (mState == STATE_SENSING) {
474              if (mNextIdlePendingDelay < System.currentTimeMillis())
475                  return;
476          }
477          ...
478      }
479
480      synchronized (DeviceIdleController.this) {
481          ...
482          if (mState == STATE_SENSING) {
483              if (mNextIdlePendingDelay < System.currentTimeMillis())
484                  return;
485          }
486          ...
487      }
488
489      synchronized (DeviceIdleController.this) {
490          ...
491          if (mState == STATE_SENSING) {
492              if (mNextIdlePendingDelay < System.currentTimeMillis())
493                  return;
494          }
495          ...
496      }
497
498      synchronized (DeviceIdleController.this) {
499          ...
500          if (mState == STATE_SENSING) {
501              if (mNextIdlePendingDelay < System.currentTimeMillis())
502                  return;
503          }
504          ...
505      }
506
507      synchronized (DeviceIdleController.this) {
508          ...
509          if (mState == STATE_SENSING) {
510              if (mNextIdlePendingDelay < System.currentTimeMillis())
511                  return;
512          }
513          ...
514      }
515
516      synchronized (DeviceIdleController.this) {
517          ...
518          if (mState == STATE_SENSING) {
519              if (mNextIdlePendingDelay < System.currentTimeMillis())
520                  return;
521          }
522          ...
523      }
524
525      synchronized (DeviceIdleController.this) {
526          ...
527          if (mState == STATE_SENSING) {
528              if (mNextIdlePendingDelay < System.currentTimeMillis())
529                  return;
530          }
531          ...
532      }
533
534      synchronized (DeviceIdleController.this) {
535          ...
536          if (mState == STATE_SENSING) {
537              if (mNextIdlePendingDelay < System.currentTimeMillis())
538                  return;
539          }
540          ...
541      }
542
543      synchronized (DeviceIdleController.this) {
544          ...
545          if (mState == STATE_SENSING) {
546              if (mNextIdlePendingDelay < System.currentTimeMillis())
547                  return;
548          }
549          ...
550      }
551
552      synchronized (DeviceIdleController.this) {
553          ...
554          if (mState == STATE_SENSING) {
555              if (mNextIdlePendingDelay < System.currentTimeMillis())
556                  return;
557          }
558          ...
559      }
560
561      synchronized (DeviceIdleController.this) {
562          ...
563          if (mState == STATE_SENSING) {
564              if (mNextIdlePendingDelay < System.currentTimeMillis())
565                  return;
566          }
567          ...
568      }
569
570      synchronized (DeviceIdleController.this) {
571          ...
572          if (mState == STATE_SENSING) {
573              if (mNextIdlePendingDelay < System.currentTimeMillis())
574                  return;
575          }
576          ...
577      }
578
579      synchronized (DeviceIdleController.this) {
580          ...
581          if (mState == STATE_SENSING) {
582              if (mNextIdlePendingDelay < System.currentTimeMillis())
583                  return;
584          }
585          ...
586      }
587
588      synchronized (DeviceIdleController.this) {
589          ...
590          if (mState == STATE_SENSING) {
591              if (mNextIdlePendingDelay < System.currentTimeMillis())
592                  return;
593          }
594          ...
595      }
596
597      synchronized (DeviceIdleController.this) {
598          ...
599          if (mState == STATE_SENSING) {
600              if (mNextIdlePendingDelay < System.currentTimeMillis())
601                  return;
602          }
603          ...
604      }
605
606      synchronized (DeviceIdleController.this) {
607          ...
608          if (mState == STATE_SENSING) {
609              if (mNextIdlePendingDelay < System.currentTimeMillis())
610                  return;
611          }
612          ...
613      }
614
615      synchronized (DeviceIdleController.this) {
616          ...
617          if (mState == STATE_SENSING) {
618              if (mNextIdlePendingDelay < System.currentTimeMillis())
619                  return;
620          }
621          ...
622      }
623
624      synchronized (DeviceIdleController.this) {
625          ...
626          if (mState == STATE_SENSING) {
627              if (mNextIdlePendingDelay < System.currentTimeMillis())
628                  return;
629          }
630          ...
631      }
632
633      synchronized (DeviceIdleController.this) {
634          ...
635          if (mState == STATE_SENSING) {
636              if (mNextIdlePendingDelay < System.currentTimeMillis())
637                  return;
638          }
639          ...
640      }
641
642      synchronized (DeviceIdleController.this) {
643          ...
644          if (mState == STATE_SENSING) {
645              if (mNextIdlePendingDelay < System.currentTimeMillis())
646                  return;
647          }
648          ...
649      }
650
651      synchronized (DeviceIdleController.this) {
652          ...
653          if (mState == STATE_SENSING) {
654              if (mNextIdlePendingDelay < System.currentTimeMillis())
655                  return;
656          }
657          ...
658      }
659
660      synchronized (DeviceIdleController.this) {
661          ...
662          if (mState == STATE_SENSING) {
663              if (mNextIdlePendingDelay < System.currentTimeMillis())
664                  return;
665          }
666          ...
667      }
668
669      synchronized (DeviceIdleController.this) {
670          ...
671          if (mState == STATE_SENSING) {
672              if (mNextIdlePendingDelay < System.currentTimeMillis())
673                  return;
674          }
675          ...
676      }
677
678      synchronized (DeviceIdleController.this) {
679          ...
680          if (mState == STATE_SENSING) {
681              if (mNextIdlePendingDelay < System.currentTimeMillis())
682                  return;
683          }
684          ...
685      }
686
687      synchronized (DeviceIdleController.this) {
688          ...
689          if (mState == STATE_SENSING) {
690              if (mNextIdlePendingDelay < System.currentTimeMillis())
691                  return;
692          }
693          ...
694      }
695
696      synchronized (DeviceIdleController.this) {
697          ...
698          if (mState == STATE_SENSING) {
699              if (mNextIdlePendingDelay < System.currentTimeMillis())
700                  return;
701          }
702          ...
703      }
704
705      synchronized (DeviceIdleController.this) {
706          ...
707          if (mState == STATE_SENSING) {
708              if (mNextIdlePendingDelay < System.currentTimeMillis())
709                  return;
710          }
711          ...
712      }
713
714      synchronized (DeviceIdleController.this) {
715          ...
716          if (mState == STATE_SENSING) {
717              if (mNextIdlePendingDelay < System.currentTimeMillis())
718                  return;
719          }
720          ...
721      }
722
723      synchronized (DeviceIdleController.this) {
724          ...
725          if (mState == STATE_SENSING) {
726              if (mNextIdlePendingDelay < System.currentTimeMillis())
727                  return;
728          }
729          ...
730      }
731
732      synchronized (DeviceIdleController.this) {
733          ...
734          if (mState == STATE_SENSING) {
735              if (mNextIdlePendingDelay < System.currentTimeMillis())
736                  return;
737          }
738          ...
739      }
740
741      synchronized (DeviceIdleController.this) {
742          ...
743          if (mState == STATE_SENSING) {
744              if (mNextIdlePendingDelay < System.currentTimeMillis())
745                  return;
746          }
747          ...
748      }
749
750      synchronized (DeviceIdleController.this) {
751          ...
752          if (mState == STATE_SENSING) {
753              if (mNextIdlePendingDelay < System.currentTimeMillis())
754                  return;
755          }
756          ...
757      }
758
759      synchronized (DeviceIdleController.this) {
760          ...
761          if (mState == STATE_SENSING) {
762              if (mNextIdlePendingDelay < System.currentTimeMillis())
763                  return;
764          }
765          ...
766      }
767
768      synchronized (DeviceIdleController.this) {
769          ...
770          if (mState == STATE_SENSING) {
771              if (mNextIdlePendingDelay < System.currentTimeMillis())
772                  return;
773          }
774          ...
775      }
776
777      synchronized (DeviceIdleController.this) {
778          ...
779          if (mState == STATE_SENSING) {
780              if (mNextIdlePendingDelay < System.currentTimeMillis())
781                  return;
782          }
783          ...
784      }
785
786      synchronized (DeviceIdleController.this) {
787          ...
788          if (mState == STATE_SENSING) {
789              if (mNextIdlePendingDelay < System.currentTimeMillis())
790                  return;
791          }
792          ...
793      }
794
795      synchronized (DeviceIdleController.this) {
796          ...
797          if (mState == STATE_SENSING) {
798              if (mNextIdlePendingDelay < System.currentTimeMillis())
799                  return;
800          }
801          ...
802      }
803
804      synchronized (DeviceIdleController.this) {
805          ...
806          if (mState == STATE_SENSING) {
807              if (mNextIdlePendingDelay < System.currentTimeMillis())
808                  return;
809          }
810          ...
811      }
812
813      synchronized (DeviceIdleController.this) {
814          ...
815          if (mState == STATE_SENSING) {
816              if (mNextIdlePendingDelay < System.currentTimeMillis())
817                  return;
818          }
819          ...
820      }
821
822      synchronized (DeviceIdleController.this) {
823          ...
824          if (mState == STATE_SENSING) {
825              if (mNextIdlePendingDelay < System.currentTimeMillis())
826                  return;
827          }
828          ...
829      }
830
831      synchronized (DeviceIdleController.this) {
832          ...
833          if (mState == STATE_SENSING) {
834              if (mNextIdlePendingDelay < System.currentTimeMillis())
835                  return;
836          }
837          ...
838      }
839
840      synchronized (DeviceIdleController.this) {
841          ...
842          if (mState == STATE_SENSING) {
843              if (mNextIdlePendingDelay < System.currentTimeMillis())
844                  return;
845          }
846          ...
847      }
848
849      synchronized (DeviceIdleController.this) {
850          ...
851          if (mState == STATE_SENSING) {
852              if (mNextIdlePendingDelay < System.currentTimeMillis())
853                  return;
854          }
855          ...
856      }
857
858      synchronized (DeviceIdleController.this) {
859          ...
860          if (mState == STATE_SENSING) {
861              if (mNextIdlePendingDelay < System.currentTimeMillis())
862                  return;
863          }
864          ...
865      }
866
867      synchronized (DeviceIdleController.this) {
868          ...
869          if (mState == STATE_SENSING) {
870              if (mNextIdlePendingDelay < System.currentTimeMillis())
871                  return;
872          }
873          ...
874      }
875
876      synchronized (DeviceIdleController.this) {
877          ...
878          if (mState == STATE_SENSING) {
879              if (mNextIdlePendingDelay < System.currentTimeMillis())
880                  return;
881          }
882          ...
883      }
884
885      synchronized (DeviceIdleController.this) {
886          ...
887          if (mState == STATE_SENSING) {
888              if (mNextIdlePendingDelay < System.currentTimeMillis())
889                  return;
890          }
891          ...
892      }
893
894      synchronized (DeviceIdleController.this) {
895          ...
896          if (mState == STATE_SENSING) {
897              if (mNextIdlePendingDelay < System.currentTimeMillis())
898                  return;
899          }
900          ...
901      }
902
903      synchronized (DeviceIdleController.this) {
904          ...
905          if (mState == STATE_SENSING) {
906              if (mNextIdlePendingDelay < System.currentTimeMillis())
907                  return;
908          }
909          ...
910      }
911
912      synchronized (DeviceIdleController.this) {
913          ...
914          if (mState == STATE_SENSING) {
915              if (mNextIdlePendingDelay < System.currentTimeMillis())
916                  return;
917          }
918          ...
919      }
920
921      synchronized (DeviceIdleController.this) {
922          ...
923          if (mState == STATE_SENSING) {
924              if (mNextIdlePendingDelay < System.currentTimeMillis())
925                  return;
926          }
927          ...
928      }
929
930      synchronized (DeviceIdleController.this) {
931          ...
932          if (mState == STATE_SENSING) {
933              if (mNextIdlePendingDelay < System.currentTimeMillis())
934                  return;
935          }
936          ...
937      }
938
939      synchronized (DeviceIdleController.this) {
940          ...
941          if (mState == STATE_SENSING) {
942              if (mNextIdlePendingDelay < System.currentTimeMillis())
943                  return;
944          }
945          ...
946      }
947
948      synchronized (DeviceIdleController.this) {
949          ...
950          if (mState == STATE_SENSING) {
951              if (mNextIdlePendingDelay < System.currentTimeMillis())
952                  return;
953          }
954          ...
955      }
956
957      synchronized (DeviceIdleController.this) {
958          ...
959          if (mState == STATE_SENSING) {
960              if (mNextIdlePendingDelay < System.currentTimeMillis())
961                  return;
962          }
963          ...
964      }
965
966      synchronized (DeviceIdleController.this) {
967          ...
968          if (mState == STATE_SENSING) {
969              if (mNextIdlePendingDelay < System.currentTimeMillis())
970                  return;
971          }
972          ...
973      }
974
975      synchronized (DeviceIdleController.this) {
976          ...
977          if (mState == STATE_SENSING) {
978              if (mNextIdlePendingDelay < System.currentTimeMillis())
979                  return;
980          }
981          ...
982      }
983
984      synchronized (DeviceIdleController.this) {
985          ...
986          if (mState == STATE_SENSING) {
987              if (mNextIdlePendingDelay < System.currentTimeMillis())
988                  return;
989          }
990          ...
991      }
992
993      synchronized (DeviceIdleController.this) {
994          ...
995          if (mState == STATE_SENSING) {
996              if (mNextIdlePendingDelay < System.currentTimeMillis())
997                  return;
998          }
999          ...
1000      }
1001
1002      synchronized (DeviceIdleController.this) {
1003          ...
1004          if (mState == STATE_SENSING) {
1005              if (mNextIdlePendingDelay < System.currentTimeMillis())
1006                  return;
1007          }
1008          ...
1009      }
1010
1011      synchronized (DeviceIdleController.this) {
1012          ...
1013          if (mState == STATE_SENSING) {
1014              if (mNextIdlePendingDelay < System.currentTimeMillis())
1015                  return;
1016          }
1017          ...
1018      }
1019
1020      synchronized (DeviceIdleController.this) {
1021          ...
1022          if (mState == STATE_SENSING) {
1023              if (mNextIdlePendingDelay < System.currentTimeMillis())
1024                  return;
1025          }
1026          ...
1027      }
1028
1029      synchronized (DeviceIdleController.this) {
1030          ...
1031          if (mState == STATE_SENSING) {
1032              if (mNextIdlePendingDelay < System.currentTimeMillis())
1033                  return;
1034          }
1035          ...
1036      }
1037
1038      synchronized (DeviceIdleController.this) {
1039          ...
1040          if (mState == STATE_SENSING) {
1041              if (mNextIdlePendingDelay < System.currentTimeMillis())
1042                  return;
1043          }
1044          ...
1045      }
1046
1047      synchronized (DeviceIdleController.this) {
1048          ...
1049          if (mState == STATE_SENSING) {
1050              if (mNextIdlePendingDelay < System.currentTimeMillis())
1051                  return;
1052          }
1053          ...
1054      }
1055
1056      synchronized (DeviceIdleController.this) {
1057          ...
1058          if (mState == STATE_SENSING) {
1059              if (mNextIdlePendingDelay < System.currentTimeMillis())
1060                  return;
1061          }
1062          ...
1063      }
1064
1065      synchronized (DeviceIdleController.this) {
1066          ...
1067          if (mState == STATE_SENSING) {
1068              if (mNextIdlePendingDelay < System.currentTimeMillis())
1069                  return;
1070          }
1071          ...
1072      }
1073
1074      synchronized (DeviceIdleController.this) {
1075          ...
1076          if (mState == STATE_SENSING) {
1077              if (mNextIdlePendingDelay < System.currentTimeMillis())
1078                  return;
1079          }
1080          ...
1081      }
1082
1083      synchronized (DeviceIdleController.this) {
1084          ...
1085          if (mState == STATE_SENSING) {
1086              if (mNextIdlePendingDelay < System.currentTimeMillis())
1087                  return;
1088          }
1089          ...
1090      }
1091
1092      synchronized (DeviceIdleController.this) {
1093          ...
1094          if (mState == STATE_SENSING) {
1095              if (mNextIdlePendingDelay < System.currentTimeMillis())
1096                  return;
1097          }
1098          ...
1099      }
1100
1101      synchronized (DeviceIdleController.this) {
1102          ...
1103          if (mState == STATE_SENSING) {
1104              if (mNextIdlePendingDelay < System.currentTimeMillis())
1105                  return;
1106          }
1107          ...
1108      }
1109
1110      synchronized (DeviceIdleController.this) {
1111          ...
1112          if (mState == STATE_SENSING) {
1113              if (mNextIdlePendingDelay < System.currentTimeMillis())
1114                  return;
1115          }
1116          ...
1117      }
1118
1119      synchronized (DeviceIdleController.this) {
1120          ...
1121          if (mState == STATE_SENSING) {
1122              if (mNextIdlePendingDelay < System.currentTimeMillis())
1123                  return;
1124          }
1125          ...
1126      }
1127
1128      synchronized (DeviceIdleController.this) {
1129          ...
1130          if (mState == STATE_SENSING) {
1131              if (mNextIdlePendingDelay < System.currentTimeMillis())
1132                  return;
1133          }
1134          ...
1135      }
1136
1137      synchronized (DeviceIdleController.this) {
1138          ...
1139          if (mState == STATE_SENSING) {
1140              if (mNextIdlePendingDelay < System.currentTimeMillis())
1141                  return;
1142          }
1143          ...
1144      }
1145
1146      synchronized (DeviceIdleController.this) {
1147          ...
1148          if (mState == STATE_SENSING) {
1149              if (mNextIdlePendingDelay < System.currentTimeMillis())
1150                  return;
1151          }
1152          ...
1153      }
1154
1155      synchronized (DeviceIdleController.this) {
1156          ...
1157          if (mState == STATE_SENSING) {
1158              if (mNextIdlePendingDelay < System.currentTimeMillis())
1159                  return;
1160          }
1161          ...
1162      }
1163
1164      synchronized (DeviceIdleController.this) {
1165          ...
1166          if (mState == STATE_SENSING) {
1167              if (mNextIdlePendingDelay < System.currentTimeMillis())
1168                  return;
1169          }
1170          ...
1171      }
1172
1173      synchronized (DeviceIdleController.this) {
1174          ...
1175          if (mState == STATE_SENSING) {
1176              if (mNextIdlePendingDelay < System.currentTimeMillis())
1177                  return;
1178          }
1179          ...
1180      }
1181
1182      synchronized (DeviceIdleController.this) {
1183          ...
1184          if (mState == STATE_SENSING) {
1185              if (mNextIdlePendingDelay < System.currentTimeMillis())
1186                  return;
1187          }
1188          ...
1189      }
1190
1191      synchronized (DeviceIdleController.this) {
1192          ...
1193          if (mState == STATE_SENSING) {
1194              if (mNextIdlePendingDelay < System.currentTimeMillis())
1195                  return;
1196          }
1197          ...
1198      }
1199
1200      synchronized (DeviceIdleController.this) {
1201          ...
1202          if (mState == STATE_SENSING) {
1203              if (mNextIdlePendingDelay < System.currentTimeMillis())
1204                  return;
1205          }
1206          ...
1207      }
1208
1209      synchronized (DeviceIdleController.this) {
1210          ...
1211          if (mState == STATE_SENSING) {
1212              if (mNextIdlePendingDelay < System.currentTimeMillis())
1213                  return;
1214          }
1215          ...
1216      }
1217
1218      synchronized (DeviceIdleController.this) {
1219          ...
1220          if (mState == STATE_SENSING) {
1221              if (mNextIdlePendingDelay < System.currentTimeMillis())
1222                  return;
1223          }
1224          ...
1225      }
1226
1227      synchronized (DeviceIdleController.this) {
1228          ...
1229          if (mState == STATE_SENSING) {
1230              if (mNextIdlePendingDelay < System.currentTimeMillis())
1231                  return;
1232          }
1233          ...
1234      }
1235
1236      synchronized (DeviceIdleController.this) {
1237          ...
1238          if (mState == STATE_SENSING) {
1239              if (mNextIdlePendingDelay < System.currentTimeMillis())
1240                  return;
1241          }
1242          ...
1243      }
1244
1245      synchronized (DeviceIdleController.this) {
1246          ...
1247          if (mState == STATE_SENSING) {
1248              if (mNextIdlePendingDelay < System.currentTimeMillis())
1249                  return;
1250          }
1251          ...
1252      }
1253
1254      synchronized (DeviceIdleController.this) {
1255          ...
1256          if (mState == STATE_SENSING) {
1257              if (mNextIdlePendingDelay < System.currentTimeMillis())
1258                  return;
1259          }
1260          ...
1261      }
1262
1263      synchronized (DeviceIdleController.this) {
1264          ...
1265          if (mState == STATE_SENSING) {
1266              if (mNextIdlePendingDelay < System.currentTimeMillis())
1267                  return;
1268          }
1269          ...
1270      }
1271
1272      synchronized (DeviceIdleController.this) {
1273          ...
1274          if (mState == STATE_SENSING) {
1275              if (mNextIdlePendingDelay < System.currentTimeMillis())
1276                  return;
1277          }
1278          ...
1279      }
1280
1281      synchronized (DeviceIdleController.this) {
1282          ...
1283          if (mState == STATE_SENSING) {
1284              if (mNextIdlePendingDelay < System.currentTimeMillis())
1285                  return;
1286          }
1287          ...
1288      }
1289
1290      synchronized (DeviceIdleController.this) {
1291          ...
1292          if (mState == STATE_SENSING) {
1293              if (mNextIdlePendingDelay < System.currentTimeMillis())
1294                  return;
1295          }
1296          ...
1297      }
1298
1299      synchronized (DeviceIdleController.this) {
1300          ...
1301          if (mState == STATE_SENSING) {
1302              if (mNextIdlePendingDelay < System.currentTimeMillis())
1303                  return;
1304          }
1305          ...
1306      }
1307
1308      synchronized (DeviceIdleController.this) {
1309          ...
1310          if (mState == STATE_SENSING) {
1311              if (mNextIdlePendingDelay < System.currentTimeMillis())
1312                  return;
1313          }
1314          ...
1315      }
1316
1317      synchronized (DeviceIdleController.this) {
1318          ...
1319          if (mState == STATE_SENSING) {
1320              if (mNextIdlePendingDelay < System.currentTimeMillis())
1321                  return;
1322          }
1323          ...
1324      }
1325
1326      synchronized (DeviceIdleController.this) {
1327          ...
1328          if (mState == STATE_SENSING) {
1329              if (mNextIdlePendingDelay < System.currentTimeMillis())
1330                  return;
1331          }
1332          ...
1333      }
1334
1335      synchronized (DeviceIdleController.this) {
1336          ...
1337          if (mState == STATE_SENSING) {
1338              if (mNextIdlePendingDelay < System.currentTimeMillis())
1339                  return;
1340          }
1341          ...
1342      }
1343
1344      synchronized (DeviceIdleController.this) {
1345          ...
1346          if (mState == STATE_SENSING) {
1347              if (mNextIdlePendingDelay < System.currentTimeMillis())
1348                  return;
1349          }
1350          ...
1351      }
1352
1353      synchronized (DeviceIdleController.this) {
1354          ...
1355          if (mState == STATE_SENSING) {
1356              if (mNextIdlePendingDelay < System.currentTimeMillis())
1357                  return;
1358          }
1359          ...
1360      }
1361
1362      synchronized (DeviceIdleController.this) {
1363          ...
1364          if (mState == STATE_SENSING) {
1365              if (mNextIdlePendingDelay < System.currentTimeMillis())
1366                  return;
1367          }
1368          ...
1369      }
1370
1371      synchronized (DeviceIdleController.this) {
1372          ...
1373          if (mState == STATE_SENSING) {
1374              if (mNextIdlePendingDelay < System.currentTimeMillis())
1375                  return;
1376          }
1377          ...
1378      }
1379
1380      synchronized (DeviceIdleController.this) {
1381          ...
1382          if (mState == STATE_SENSING) {
1383              if (mNextIdlePendingDelay < System.currentTimeMillis())
1384                  return;
1385          }
1386          ...
1387      }
1388
1389      synchronized (DeviceIdleController.this) {
1390          ...
1391          if (mState == STATE_SENSING) {
1392              if (mNextIdlePendingDelay < System.currentTimeMillis())
1393                  return;
1394          }
1395          ...
1396      }
1397
1398      synchronized (DeviceIdleController.this) {
1399          ...
1400          if (mState == STATE_SENSING) {
1401              if (mNextIdlePendingDelay < System.currentTimeMillis())
1402                  return;
1403          }
1404          ...
1405      }
1406
1407      synchronized (DeviceIdleController.this) {
1408          ...
1409          if (mState == STATE_SENSING) {
1410              if (mNextIdlePendingDelay < System.currentTimeMillis())
1411                  return;
1412          }
1413          ...
1414      }
1415
1416      synchronized (DeviceIdleController.this) {
1417          ...
1418          if (mState == STATE_SENSING) {
1419              if (mNextIdlePendingDelay < System.currentTimeMillis())
1420                  return;
1421          }
1422          ...
1423      }
1424
1425      synchronized (DeviceIdleController.this) {
1426          ...
1427          if (mState == STATE_SENSING) {
1428              if (mNextIdlePendingDelay < System.currentTimeMillis())
1429                  return;
1430          }
1431          ...
1432      }
1433
1434      synchronized (DeviceIdleController.this) {
1435          ...
1436          if (mState == STATE_SENSING) {
1437              if (mNextIdlePendingDelay < System.currentTimeMillis())
1438                  return;
1439          }
1440          ...
1441      }
1442
1443      synchronized (DeviceIdleController.this) {
1444          ...
1445          if (mState == STATE_SENSING) {
1446              if (mNextIdlePendingDelay < System.currentTimeMillis())
1447                  return;
1448          }
1449          ...
1450      }
1451
1452      synchronized (DeviceIdleController.this) {
1453          ...
1454          if (mState == STATE_SENSING) {
1455              if (mNextIdlePendingDelay < System.currentTimeMillis())
1456                  return;
1457          }
1458          ...
1459      }
1460
1461      synchronized (DeviceIdleController.this) {
1462          ...
1463          if (mState == STATE_SENSING) {
1464              if (mNextIdlePendingDelay < System.currentTimeMillis())
1465                  return;
1466          }
1467          ...
1468      }
1469
1470      synchronized (DeviceIdleController.this) {
1471          ...
1472          if (mState == STATE_SENSING) {
1473              if (mNextIdlePendingDelay < System.currentTimeMillis())
1474                  return;
1475          }
1476          ...
1477      }
1478
1479      synchronized (DeviceIdleController.this) {
1480          ...
1481          if (mState == STATE_SENSING) {
1482              if (mNextIdlePendingDelay < System.currentTimeMillis())
1483                  return;
1484          }
1485          ...
1486      }
1487
1488      synchronized (DeviceIdleController.this) {
1489          ...
1490          if (mState == STATE_SENSING) {
1491              if (mNextIdlePendingDelay < System.currentTimeMillis())
1492                  return;
1493          }
1494          ...
1495      }
1496
1497      synchronized (DeviceIdleController.this) {
1498          ...
1499          if (mState == STATE_SENSING) {
1500              if (mNextIdlePendingDelay < System.currentTimeMillis())
1501                  return;
1502          }
1503          ...
1504      }
1505
1506      synchronized (DeviceIdleController.this) {
1507          ...
1508          if (mState == STATE_SENSING) {
1509              if (mNextIdlePendingDelay < System.currentTimeMillis())
1510                  return;
1511          }
1512          ...
1513      }
1514
1515      synchronized (DeviceIdleController.this) {
1516          ...
1517          if (mState == STATE_SENSING) {
1518              if (mNextIdlePendingDelay < System.currentTimeMillis())
1519                  return;
1520          }
1521          ...
1522      }
1523
1524      synchronized (DeviceIdleController.this) {
1525          ...
1526          if (mState == STATE_SENSING) {
1527              if (mNextIdlePendingDelay < System.currentTimeMillis())
1528                  return;
1529          }
1530          ...
1531      }
1532
1533      synchronized (DeviceIdleController.this) {
1534          ...
1535          if (mState == STATE_SENSING) {
1536              if (mNextIdlePendingDelay < System.currentTimeMillis())
1537                  return;
1538          }
1539          ...
1540      }
1541
1542      synchronized (DeviceIdleController.this) {
1543          ...
1544          if (mState == STATE_SENSING) {
1545              if (mNextIdlePendingDelay < System.currentTimeMillis())
1546                  return;
1547          }
1548          ...
1549      }
1550
1551      synchronized (DeviceIdleController.this) {
1552          ...
1553          if (mState == STATE_SENSING) {
1554              if (mNextIdlePendingDelay < System.currentTimeMillis())
1555                  return;
1556          }
1557          ...
1558      }
1559
1560      synchronized (DeviceIdleController.this) {
1561          ...
1562          if (mState == STATE_SENSING) {
1563              if (mNextIdlePendingDelay < System.currentTimeMillis())
1564                  return;
1565          }
1566          ...
1567      }
1568
1569      synchronized (DeviceIdleController.this) {
1570          ...
1571          if (mState == STATE_SENSING) {
1572              if (mNextIdlePendingDelay < System.currentTimeMillis())
1573                  return;
1574          }
1575          ...
1576      }
1577
1578      synchronized (DeviceIdleController.this) {
1579          ...
1580          if (mState == STATE_SENSING) {
1581              if (mNextIdlePendingDelay < System.currentTimeMillis())
1582                  return;
1583          }
1584         
```

components of the application. Finally, we will describe the main techniques examined in this paper, namely ICC (Inter-Component Communication) and asynchronous programming.

### 3.1 Android system infrastructure

Android is an operating system for mobile devices such as smartphones and tablet computers, which is developed by the Open Handset Alliance led by Google. Android has evolved quickly since its first commercial version Android 1.0 that was released on September 2008. The newest version, Android 7 whose code name is Nougat was released on August 22, 2016.

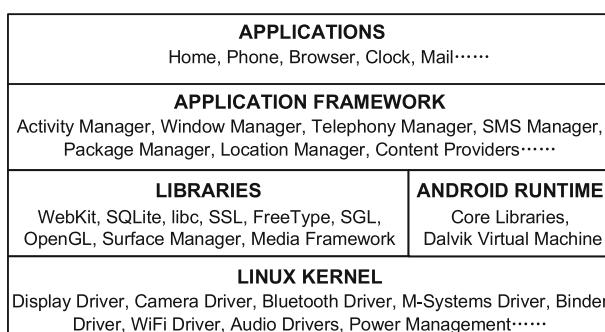
Although Android evolves quickly, its infrastructure keeps stable, as shown in Fig. 2. Android consists of five parts: Linux Kernel, Android Runtime, Libraries, Application Framework, and Applications. Linux Kernel manages hardware drivers, network, battery, system security, memory, etc. Android Runtime consists of Core Libraries which provide most functionalities in Java core libraries, and a Dalvik virtual machine (DVM). Android can run multiple DVMs simultaneously, with one application in each DVM.

Application Framework provides a set of services (e.g., Resource Manager, Notification Manager, Activity Manager) for applications, so programmers can develop varied applications by invoking framework's APIs. Applications is the top layer of Android which hosts built-in applications and third-party applications. The layered infrastructure ensures that the lower layers provide services for higher layers, and also benefits programmers for different layers concentrating on their own layers.

### 3.2 Android application structure

An Android application has at least one of the following components: Activity, Service, Broadcast Receiver, and Content Provider. An activity is the entry point for interacting with the user. It represents a single screen with a user interface. A service is a general-purpose entry point for keeping an application running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface.

A broadcast receiver provides a new approach to the Android system for sending events to the app, and empowers the app to receive the announcements broadcasted by the system. Because broadcast receivers are another well-defined entry into the application, the system



**Fig. 2** Infrastructure of Android system

can deliver broadcasts even to applications that aren't currently running. A content provider manages a shared set of application data that you can store in the file system, in an **SQLite** database, on the web, or on any other persistent storage location that your application can access. Through the content provider, other applications can query or modify the data if the content provider allows it.

### 3.3 ICC

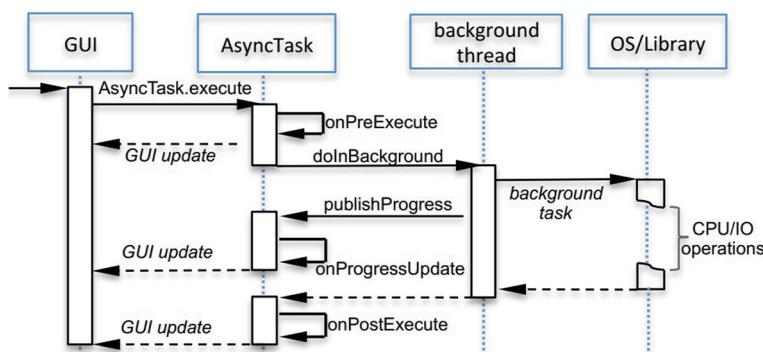
Different components in an application can communicate using ICC objects, mainly Intents. By the same way, components can also communicate across applications, allowing developers to reuse functionality. For example, Google Map provides navigation function, so any restaurant applications just need to give the location coordinates and invoke Google Map for navigation by sending appropriate intent. Android intents are two types in nature.

- Explicit intents, explicitly define the exact component which should be called by the Android system, by using the Java class as identifier. Explicit intents are often used in ICC within the application because the name of invoked should be given correctly.
- Implicit intents specify the action which should be performed by other components or applications. Implicit intents are usually used for IAC (Inter-Application Communication) since the action should be performed rather than the exact name of the called component should be specified.

### 3.4 Android asynchronous programming

To ease asynchronous programming which is a widely used approach to reduce application latency, Android provides three major async constructs: `AsyncTask`, `IntentService`, and `AsyncTaskLoader`.

- `AsyncTask` provides a `doInBackground` method for encapsulating asynchronous work. Besides, it provides four event handlers (i.e., `onPreExecute`, `onProgressUpdate`, `onPostExecute`, and `OnCancel`) which are run in the UI thread. The `doInBackground` and these event handlers share variables through which the background task can communicate with UI (Lin et al. 2015). Figure 3 depicts the workflow of `AsyncTask`. `AsyncTasks` should ideally be used for short operations (a few seconds at the most).



**Fig. 3** The workflow of `AsyncTask` (Lin et al. 2015)

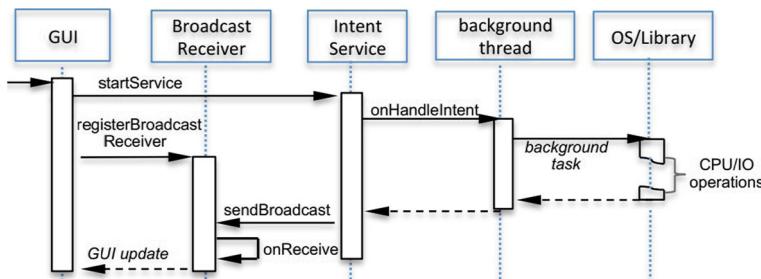
- IntentService is a base class for Services that handle asynchronous requests (expressed as Intents) on demand (IntentService <https://developer.android.com/reference/android/app/IntentService.html>). Clients send requests through startService (intent) calls; the service is started as needed, handles each intent in turn using a worker thread, and stops itself when it runs out of work. Please note that all requests are handled on a single worker thread—they may take as long as necessary and will not block the application’s main loop (IntentService <https://developer.android.com/reference/android/app/IntentService.html>). To get the task result, the GUI that starts the service should register a broadcast receiver. After the task is finished, IntentService sends its task result via the sendBroadcast method. Once the registered receiver on GUI receives this broadcast, its onReceive method will be executed on UI thread, so it can get the task result and update GUI (Lin et al. 2015). The workflow of IntentService is given in Fig. 4. Therefore, IntentService is a good choice for long-running tasks.
- AsyncTaskLoader, as its name suggests, is built on top of AsyncTask, and it provides similar handlers as AsyncTask. Unlike AsyncTask, AsyncTaskLoader is lifecycle aware: Android system binds/unbinds the background task with GUI according to GUI’s lifecycle (Lin et al. 2015).

## 4 Unprotected intents

In this paper, we examine Android 6 because it accounts for 31.2% market share in May 2017 (Bandla <http://www.gadgetdetail.com/android-version-market-share-distribution/>) instead of the newest Android N/7 because it is installed on very few smartphones, about just 7.1% (Bandla <http://www.gadgetdetail.com/android-version-market-share-distribution/>). All experiments are conducted on a real smartphone, Huawei Nexus 6P.

We develop a tool to find all unprotected intents defined in Android 6 automatically, which consists of the following steps. First, the tool parses the source code of Android system to search for this pattern “new intent,” because all intents should be defined according to the pattern. Then, the tool searches for the types of intents by exploring the fact that Android always defines the types in the beginning of source files as constant strings. After eliminating the duplicate intent types, we get 1235 intents defined in Android 6.

Then, we determine all protected intents that should not be sent by third-party applications by analyzing the manifest file /frameworks/base/core/res/AndroidManifest.xml because Android lists all protected intents in it. We find



**Fig. 4** The workflow of IntentService (<https://developer.android.com/reference/android/app/IntentService.html>)

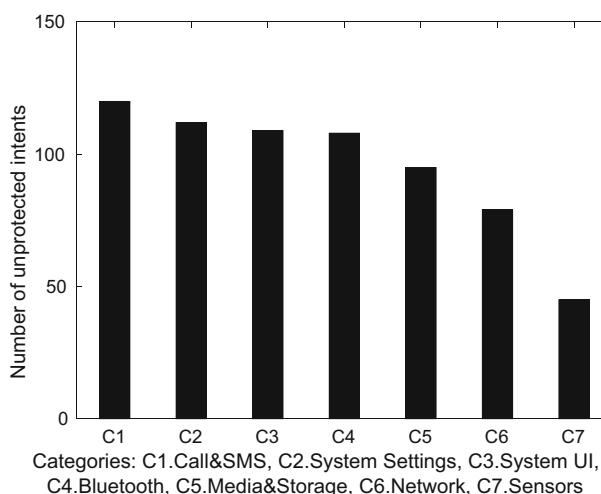
that all protected intents are defined in a fixed pattern like `<protected-broadcast android:name=XXX/>`, where XXX is a string indicating the intent type. Android forbids third-party applications to send protected intents as follows: (1) before forwarding an intent to the target component, Android system checks whether the intent is in the protected list; (2) if so, Android checks whether the application that sends the intent has system privilege; (3) if not, Android system terminates the application with a crash. Our tool parses all manifest files and extracts 261 protected intents from them, and therefore the number of unprotected intents should be 974.

We count high-risk unprotected intents that involve hardware and system operations, and classify them into seven categories, as shown in Fig. 5. For example, the intent `android.provider.Telephony.SMS_REJECTED` belongs to the Call & SMS category. One observation from Fig. 5 is that system components use unprotected intents for async operations and communications frequently, indicating that unprotected intents would be good choices to attack Android system. For instance, there are 120, 112, 109 unprotected intents belong to Call & SMS, System Settings, System UI, respectively. Sections 5 and 6 will present the attacking sceneries by exploiting the unprotected intents.

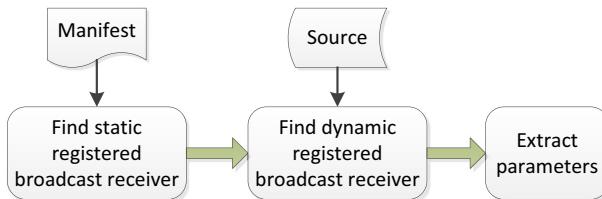
## 5 ATUIN: attacks by exploiting unprotected intents automatically

This section details the design and implementation of our tool, ATUIN, to demonstrate the feasibility of attacking CPU automatically. That is, it will result in high CPU utilization ratio, thus the responsiveness of smartphones can be weakened. The basic idea of this attack is to force Android system to repeatedly execute heavy-weight functions for handling intents by sending those intents periodically.

The proposed CPU attack is stealthy, although it is not complicated for the following reasons. First, the malware itself does not contain much code to be executed; instead, it forces Android to execute a lot of system code. Second, the malware does not need any permissions so that it can evade permission-based detection approaches. Moreover, a hacker can adjust attacking strength flexibly by setting the speed of sending intents.



**Fig. 5** Numbers of different categories of unprotected intents



**Fig. 6** Workflow of ATUIN

To launch an effective and efficient CPU attack, ATUIN aims to use the intents that force Android system to spend computational resources to handle them. Note that Android system will not process all intents. For example, the intents that are used for informing third-party applications about the change of system state will not be processed by Android system. Actually, Android system just sends those intents, rather than receiving them. If any third-party applications send the intents without processing code, Android system simply discards them.

To the end, ATUIN follows the workflow as shown in Fig. 6, which consists of three steps. The first step is finding all statically registered broadcast receivers. According to Android programming guides, all statically registered broadcast receivers should be listed in `manifest.xml`. Therefore, ATUIN parses all manifest files and extracts necessary information from them, such as which component can receive broadcasts and which types of broadcasts can be received. Fortunately, Android defines a fixed pattern to register broadcast receivers in manifest files, facilitating the parsing process of ATUIN.

Figure 7 gives a code snippet in `/packages/apps/Bluetooth/AndroidManifest.xml`. The code highlighted by blue lines indicates the keywords searched by ATUIN. For example, ATUIN searches for `<receiver>` and `</receiver>` to locate the registration of a broadcast receiver, and searches for `android:name=` to find the component that receives intents. Moreover, ATUIN looks for `<intent-filter>` and `</intent-filter>` to locate intent filters. Then, ATUIN searches for the pattern `<action android:name=` to find the type of intent that can be processed.

The code underlined by red lines contains the information ATUIN required. For example, Fig. 7 indicates that the component `.opp.BluetoothOppReceiver` can receive two types of intents, `android.bluetooth.adapter.action.STATE_CHANGED` and `android.btopp.intent.action.OPEN_RECEIVED_FILES`.

The second step is discovering dynamically registered broadcast receivers that are widely-used when developers want to control the life circle of the broadcast receivers. ATUIN finds this kind of broadcast receivers by code analysis. Note that Android enables applications to register broadcast receivers dynamically (i.e., the framework

```

104     <receiver
105         android:process="@string/process"
106         android:exported="true"
107         android:name=".opp.BluetoothOppReceiver"
108         android:enabled="@bool/profile_supported_opp">
109             <intent-filter>
110                 <action android:name="android.bluetooth.adapter.action.STATE_CHANGED" />
111                 <!--action android:name="android.intent.action.BOOT_COMPLETED" /-->
112                 <action android:name="android.btopp.intent.action.OPEN_RECEIVED_FILES" />
113             </intent-filter>
114         </receiver>
  
```

**Fig. 7** A statically registered broadcast receiver

API, `registerReceiver` should be invoked). Figure 8 illustrates how the component `GsmServiceStateTracker` registers a broadcast receiver at runtime to receive the intent `ACTION_RADIO_OFF`.

ATUIN firstly searches for the API invocation, `registerReceiver`, and then gets the second parameter, `filter` in this example. Afterwards, ATUIN looks for the API invocation, `addAction` before the invocation of `registerReceiver`, and then gets the parameter, `ACTION_RADIO_OFF`. Finally, ATUIN searches for the definition of `ACTION_RADIO_OFF` which is a constant string in Android source code.

The third step is extracting the data attached to the intent since ATUIN aims to trigger the processing code of the corresponding intent. If the data is not provided correctly, the processing logic will abort quickly, result in non-obvious attacking effect. ATUIN conducts inter-procedural data flow analysis to discover valid data parameters of intents. For a better understanding of the inter-procedural analysis, we take the code in Fig. 27 as an example. The code `getIntExtra(NfcAdapter.EXTRA_ADAPTER_STATE, NfcAdapter.STATE_OFF)` indicates that the parameter is named `EXTRA_ADAPTER_STATE` and it is an integer. Then, inter-procedural data flow analysis shows that the data attached in the intent is passed as a parameter `newState` of the function `handleNfcStateChanged`. After that, ATUIN searches for the statement that `newState` is compared with a constant integer since the comparison is used for executing the corresponding program logic for different data. Hence, ATUIN finds that the attached data, `NfcAdapter.EXTRA_ADAPTER_DATA` can be set as `NfcAdapter.STATE_OFF`, `NfcAdapter.STATE_ON`, `NfcAdapter.STATE_TURNING_OFF`, or `NfcAdapter.STATE_TURNING_ON`. According to this process, ATUIN can extract and set parameters attached to the targeted intent.

The experiments consist of three sceneries: no attacks, attack by sending 20 intents with and without processing code respectively, as shown in Figs. 9, 10, and 11. The observation is that our tool attacks CPU effectively, i.e., CPU utilization ratio rises from 11.17 to 71.13%. Moreover, to adapt to heavy workload, Android adjusts CPU frequency from 652.8 MHz to 1.22 GHz. On the contrary, attacking by sending the intents without processing code can only slightly increase CPU utilization ratio by 4.74%.

## 6 Case studies: critical vulnerabilities

In this section, we analyze the unprotected intents and examine the negative effect on Android system if they are exploited by an attacker. In particular, we investigate the processing code of selected unprotected intents in depth and find tens of critical vulnerabilities that have not been reported before. This section describes five important vulnerabilities. The attacks exploiting each vulnerability are recorded and the videos can be found at <https://goo.gl/QZn7Rk>.

Since it is time-consuming to analyze each unprotected intent manually, we prefer to the unprotected intents that either (1) change system states, (2) operate hardware component

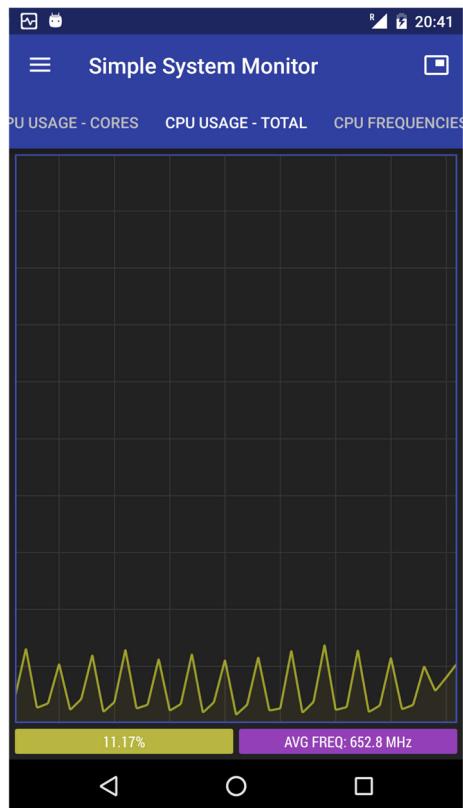
```

252     filter = new IntentFilter();
253     Context context = phone.getContext();
254     filter.addAction(ACTION_RADIO_OFF);
255     context.registerReceiver(mIntentReceiver, filter);

```

**Fig. 8** A dynamically registered broadcast receiver

**Fig. 9** CPU utilization ratio without attacks



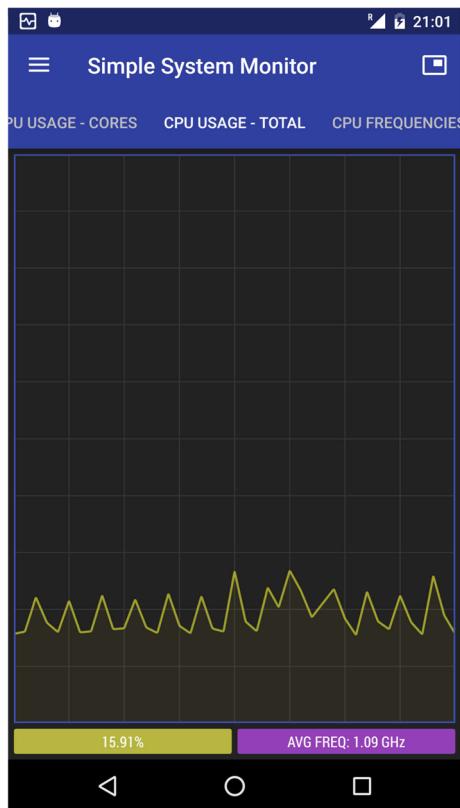
(e.g., Wi-Fi, SIM card, UI), or (3) get access to private information (e.g., contact, photos). Then, we analyze the processing code and generate attacks manually. We try four heuristic strategies to launch attacks. The first is sending an unprotected intent once with valid data. The second is sending an unprotected intent once with invalid data. The third is sending an unprotected intent with valid data repeatedly at a higher rate. The last is sending an unprotected intent with invalid data repeatedly at a higher rate.

The last step is checking whether the performance or functionalities of Android system or applications are impaired. To do so, we try each critical functionalities manually, such as Wi-Fi, Bluetooth, Telephone to examine whether they can work as usual. Moreover, we resort to behavior monitoring tool (e.g., DROIDBOX <https://github.com/pjplantz/droidbox>) to find abnormal behaviors as well as privacy leakage. Furthermore, we use performance profiling tool (e.g., Android Studio Performance Profiling Tools <https://developer.android.com/studio/profile/index.html>) to discover abnormal performance degradation, such as high CPU utilization ratio, fast power depletion, excessive memory consumption.

## 6.1 Wi-Fi DoS

This attack takes advantage of the unprotected intent: ACTION\_DEVICE\_IDLE that is defined in `/frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiController.java`. It is easy to reproduce the attack which broadcasts the intent without data attached to the intent. After the attack is successfully

**Fig. 10** CPU utilization ratio when sending 20 intents without processing code



launched, Wi-Fi signal will be blocked and Android system cannot connect to any access points, as shown in Fig. 12.

The vulnerability-related code is shown in Fig. 13. The component `WifiController` registers a broadcast receiver (Line 181), and if one `ACTION_DEVICE_IDLE` intent is received (Line 185), a message termed by `CMD_DEVICE_IDLE` will be sent. `WifiController` defines a routine, `processMessage` (Line 710) to handle all Wi-Fi related commands. In particular, if the message is `CMD_DEVICE_IDLE`, the function `checkLocksAndTransitionWhenDeviceIdle` will be invoked. In this function, we can find state transitions.

Android defines 12 states (as shown in Fig. 14) and maintains state transitions in `/frameworks/opt/net/wifi/service/java/com/android/server/wifi/WifiController.java`. The 12 states are not in the same hierarchy; instead, some states are the sub-states of another state. For instance, `addState(mApStaDisabledState, mDefaultState)` indicates that `mApStaDisabledState` is a sub-state of `mDefaultState`. The complete hierarchical relation of states is depicted in Fig. 15. We can see that `mDefaultState` is the parent state of all other states and all states involving the function `checkLocksAndTransitionWhenDeviceIdle` are the sub-states of `mDeviceInactiveState`.

Our attack implements a piece of malware without requiring permissions which sends the unprotected intent, `CMD_DEVICE_IDLE`. After processing by `checkLocksAndTransitionWhenDeviceIdle`, the parent state, `mDefaultState` will handle

**Fig. 11** CPU utilization ratio when sending 20 intents with processing code



this intent, as shown in Fig. 16. At Line 359, a global variable `mDeviceIdle` is set to `true`.

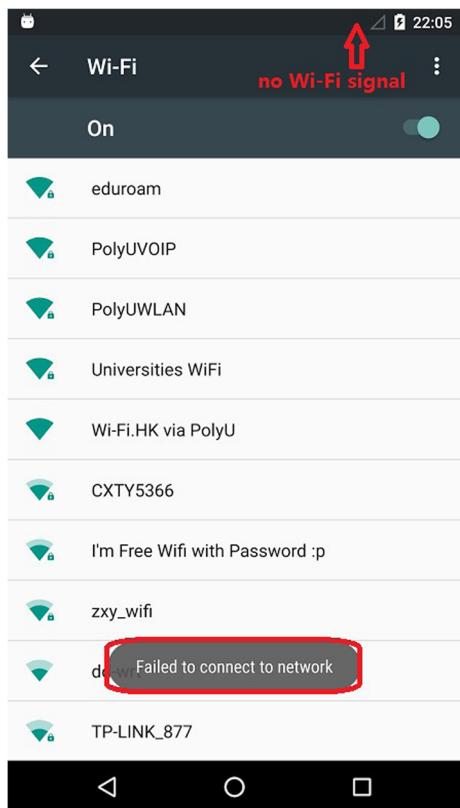
When users tag the Wi-Fi slider (as shown in Fig. 12), the component `wifiService` will generate an intent, `CMD_WIFI_TOGGLED`. However, none of the five sub-states of `mDeviceInactiveState` can handle this intent. Interestingly, `mDeviceInactiveState` cannot process this intent, and hence this intent will be forwarded to its parent state, `mStaEnabledState`. Afterwards, `mStaEnabledState` handles `CMD_WIFI_TOGGLED` as shown in Fig. 17, indicating that Android system will transfer to one of the two states, `mStaDisabledWithScanState` and `mApStaDisabledState`.

`mStaDisabledWithScanState` and `mApStaDisabledState` process `CMD_WIFI_TOGGLED` in a similar way, as shown in Fig. 18. If the variable `mDeviceIdle` is `false`, Android system will transfer to `mDeviceActiveState`, the state a smartphone can connect to access points. However, our attack makes `mDeviceIdle` be `true` by sending the unprotected intent, `ACTION_DEVICE_IDLE`. As a consequence, we successfully DoS the Wi-Fi component.

## 6.2 Telephone signal block

Our attack exploits the unprotected intent, `ACTION_RADIO_OFF` that is defined in `/frameworks/opt/telephony/src/java/com/android/internal/telephony/`

**Fig. 12** Symptom after Wi-Fi DoS attack



`ServiceStateTracker.java`. Since the definition of this intent cannot be found in Android API, it should be reserved for internal use only. However, Android 6 does not protect the intent, and thus any third-party applications can send the intent without restrictions.

After sending one `ACTION_RADIO_OFF` intent, the signal of the smartphone will be cut immediately and the signal will reappear in a short while. Therefore, by sending the intent periodically, we can block telephone signal at all, as shown in Fig. 19. The most obvious symptom is that a smartphone under attack cannot make or receive telephone calls.

The vulnerable code is located in `/frameworks/opt/telephony/src/java/com/android/internal/telephony/gsm/GsmServiceStateTracker.java` (as shown in Fig. 20). It registers a broadcast receiver (Line 171) to receive the intent (Line 183). When an `ACTION_RADIO_OFF` is received, the function `powerOffRadioSafely` is invoked (Line 186), where the function `hangupAndPowerOff` is called (Line 2153). `hangupAndPowerOff` firstly hangs up all active phone calls if any (Line 562 to 566), and then powers off the radio by invoking `setRadioPower` (Line 568). The immediate observation of our attack is that telephone signal vanishes because the radio component is turned off.

### 6.3 SIM card removal

Our attack can disable all SIM-card-related functionalities, such as making/receiving calls, sending/receiving SMS messages by periodically sending an unprotected intent,

```

180     mContext.registerReceiver(
181         new BroadcastReceiver() {
182             @Override
183             public void onReceive(Context context, Intent intent) {
184                 String action = intent.getAction();
185                 if (action.equals(ACTION_DEVICE_IDLE)) {
186                     sendMessage(CMD_DEVICE_IDLE);
187                 }
188             }
189         }
190     );
191
192     public boolean processMessage(Message msg) {
193         if (msg.what == CMD_DEVICE_IDLE) {
194             checkLocksAndTransitionWhenDeviceIdle();
195         }
196     }
197
198     private void checkLocksAndTransitionWhenDeviceIdle() {
199         if (!mLocks.hasLocks()) {
200             switch (mLocks.getStrongestLockMode()) {
201                 case WIFI_MODE_FULL:
202                     transitionTo(mFullLockHeldState);
203                     break;
204                 case WIFI_MODE_FULL_HIGH_PERF:
205                     transitionTo(mFullHighPerfLockHeldState);
206                     break;
207                 case WIFI_MODE_SCAN_ONLY:
208                     transitionTo(mScanOnlyLockHeldState);
209                     break;
210                 default:
211                     log("Illegal lock " + mLocks.getStrongestLockMode());
212             }
213         } else {
214             if (mSettingsStore.isScanAlwaysAvailable()) {
215                 transitionTo(mScanOnlyLockHeldState);
216             } else {
217                 transitionTo(mNoLockHeldState);
218             }
219         }
220     }
221 }

```

**Fig. 13** Vulnerable code exploited by Wi-Fi DoS attack

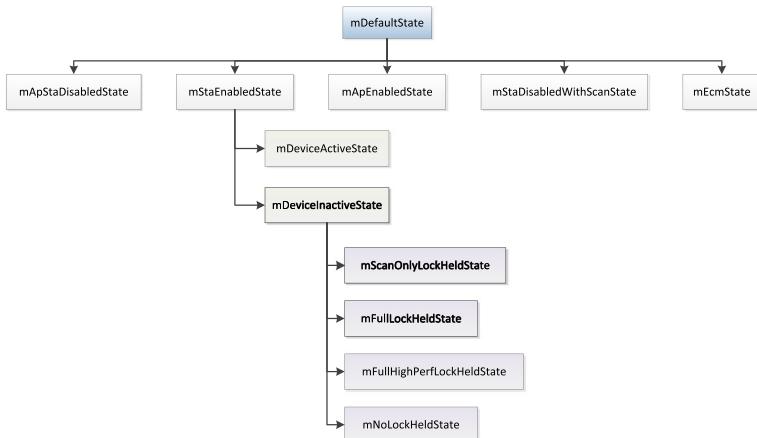
ACTION\_CARRIER\_CONFIG\_CHANGED that is defined in /frameworks/base/telephony/java/android/telephony/CarrierConfigManager.java as shown in Fig. 21. Interestingly, the explanation for the intent definition shows that this intent should be sent by the system. However, Android 6 does not protect the intent from being sent by third-party applications.

```

122     private DefaultState mDefaultState = new DefaultState();
123     private StaEnabledState mStaEnabledState = new StaEnabledState();
124     private ApStaDisabledState mApStaDisabledState = new ApStaDisabledState();
125     private StaDisabledWithScanState mStaDisabledWithScanState = new StaDisabledWithScanState();
126     private ApEnabledState mApEnabledState = new ApEnabledState();
127     private DeviceActiveState mDeviceActiveState = new DeviceActiveState();
128     private DeviceInactiveState mDeviceInactiveState = new DeviceInactiveState();
129     private ScanOnlyLockHeldState mScanOnlyLockHeldState = new ScanOnlyLockHeldState();
130     private FullLockHeldState mFullLockHeldState = new FulllockHeldState();
131     private FullHighPerfLockHeldState mFullHighPerfLockHeldState = new FullHighPerfLockHeldState();
132     private NoLockHeldState mNoLockHeldState = new NoLockHeldState();
133     private EcmState mEcmState = new EcmState();
134
135     ....
136     addState(mDefaultState);
137     addState(mApStaDisabledState, mDefaultState);
138     addState(mStaEnabledState, mDefaultState);
139     addState(mDeviceActiveState, mStaEnabledState);
140     addState(mDeviceInactiveState, mStaEnabledState);
141     addState(mScanOnlyLockHeldState, mDeviceInactiveState);
142     addState(mFullLockHeldState, mDeviceInactiveState);
143     addState(mFullHighPerfLockHeldState, mDeviceInactiveState);
144     addState(mNoLockHeldState, mDeviceInactiveState);
145     addState(mStaDisabledWithScanState, mDefaultState);
146     addState(mApEnabledState, mDefaultState);
147     addState(mEcmState, mDefaultState);

```

**Fig. 14** Definitions of twelve states of Wi-Fi

**Fig. 15** Hierarchical relation of states**Fig. 16** Processing code in `mDefaultState`

```

358     case CMD_DEVICE_IDLE:
359         mDeviceIdle = true;
360         updateBatteryWorkSource();
361         break;
  
```

```

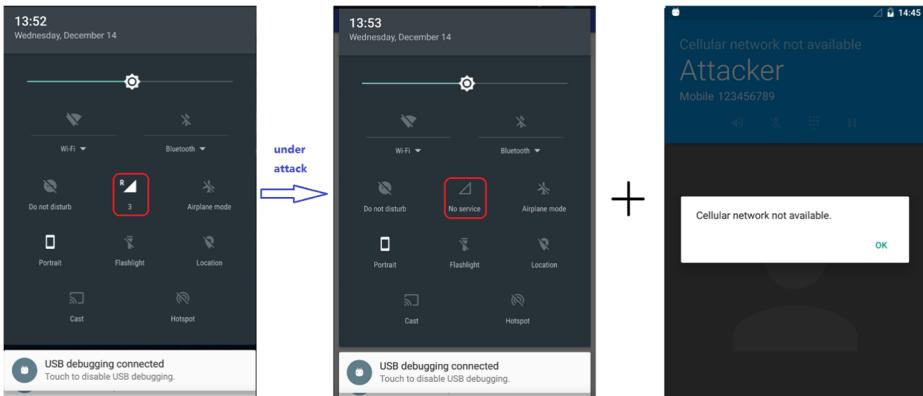
489     class StaEnabledState extends State {
490         @Override
491         public void enter() {
492             mWifiStateMachine.setSupplicantRunning(true);
493         }
494         @Override
495         public boolean processMessage(Message msg) {
496             switch (msg.what) {
497                 case CMD_WIFI_TOGGLED:
498                     if (!mSettingsStore.isWifiToggleEnabled()) {
499                         if (mSettingsStore.isScanAlwaysAvailable()) {
500                             transitionTo(mStaDisabledWithScanState);
501                         } else {
502                             transitionTo(mApStaDisabledState);
503                         }
  
```

**Fig. 17** Processing code in `mStaEnabledState`

```

554     public boolean processMessage(Message msg) {
555         switch (msg.what) {
556             case CMD_WIFI_TOGGLED:
557                 if (mSettingsStore.isWifiToggleEnabled()) {
558                     if (doDeferEnable(msg)) {
559                         if (!mHaveDeferredEnable) {
560                             // have 2 toggles now, inc serial number and ignore both
561                             mDeferredEnableSerialNumber++;
562                         }
563                         mHaveDeferredEnable = !mHaveDeferredEnable;
564                         break;
565                     }
566                     if (mDeviceIdle == false) {
567                         transitionTo(mDeviceActiveState);
568                     } else {
569                         checkLocksAndTransitionWhenDeviceIdle();
570                     }
  
```

**Fig. 18** Processing code in `mStaDisabledWithScanState`



**Fig. 19** Symptom after telephone signal block attack

The component `SimChangeReceiver` (as shown in Fig. 22) registers a broadcast receiver to receive the intent (Line 42) and processes the intent in corresponding callback function (Line 46). When an `ACTION_CARRIER_CONFIG_CHANGED` intent is received, the settings of SIM-card-related components, such as voice mail and phone account, are refreshed. By sending the unprotected intent repeatedly, the attack is capable of denying the services of SIM-card-related components. Figure 23 demonstrates that the tested smartphone can neither find the SIM card nor make calls under the attack.

#### 6.4 Homescreen hiding

Android's homescreen provides shortcuts to applications, which is an user-friendly design. Our attack is able to hide all shortcuts on the homescreen by just sending

```

560     protected void hangupAndPowerOff() {
561         // hang up all active voice calls
562         if (mPhone isInCall()) {
563             mPhone.mCT.mRingingCall.hangupIfAlive();
564             mPhone.mCT.mBackgroundCall.hangupIfAlive();
565             mPhone.mCT.mForegroundCall.hangupIfAlive();
566         }
567
568         mCi.setRadioPower(false, null);
569     }
570
571     private BroadcastReceiver mIntentReceiver = new BroadcastReceiver() {
572         @Override
573         public void onReceive(Context context, Intent intent) {
574
575             if (intent.getAction().equals(ACTION_RADIO_OFF)) {
576                 mAlarmSwitch = false;
577                 DcTrackerBase dcTracker = mPhone.mDcTracker;
578                 powerOffRadioSafely(dcTracker);
579             }
580
581             synchronized (this) {
582                 if (!mPendingRadioPowerOffAfterDataOff) {
583
584                     hangupAndPowerOff();
585                 }
586             }
587         }
588     }
589
590     public void powerOffRadioSafely(DcTrackerBase dcTracker) {
591         synchronized (this) {
592             if (!mPendingRadioPowerOffAfterDataOff) {
593
594                 hangupAndPowerOff();
595             }
596         }
597     }
598
599     protected void handleDataOff() {
600         synchronized (this) {
601             if (!mPendingRadioPowerOffAfterDataOff) {
602
603                 powerOffRadioSafely(mDcTracker);
604             }
605         }
606     }
607
608     protected void handleDataOn() {
609         synchronized (this) {
610             if (mPendingRadioPowerOffAfterDataOff) {
611
612                 powerOnRadioSafely(mDcTracker);
613             }
614         }
615     }
616
617     protected void powerOnRadioSafely(DcTrackerBase dcTracker) {
618         synchronized (this) {
619             if (mPendingRadioPowerOffAfterDataOff) {
620
621                 handleDataOn();
622             }
623         }
624     }
625
626     protected void handlePowerOn() {
627         synchronized (this) {
628             if (mPendingRadioPowerOffAfterDataOff) {
629
630                 powerOnRadioSafely(mDcTracker);
631             }
632         }
633     }
634
635     protected void powerOnRadioSafely() {
636         synchronized (this) {
637             if (mPendingRadioPowerOffAfterDataOff) {
638
639                 handlePowerOn();
640             }
641         }
642     }
643
644     protected void handlePowerOff() {
645         synchronized (this) {
646             if (!mPendingRadioPowerOffAfterDataOff) {
647
648                 powerOffRadioSafely(mDcTracker);
649             }
650         }
651     }
652
653     protected void powerOffRadioSafely() {
654         synchronized (this) {
655             if (!mPendingRadioPowerOffAfterDataOff) {
656
657                 handlePowerOff();
658             }
659         }
660     }
661
662     protected void handleDataOff() {
663         synchronized (this) {
664             if (!mPendingRadioPowerOffAfterDataOff) {
665
666                 powerOffRadioSafely(mDcTracker);
667             }
668         }
669     }
670
671     protected void powerOffRadioSafely() {
672         synchronized (this) {
673             if (!mPendingRadioPowerOffAfterDataOff) {
674
675                 handleDataOff();
676             }
677         }
678     }
679
680     protected void handlePowerOn() {
681         synchronized (this) {
682             if (mPendingRadioPowerOffAfterDataOff) {
683
684                 powerOnRadioSafely(mDcTracker);
685             }
686         }
687     }
688
689     protected void powerOnRadioSafely() {
690         synchronized (this) {
691             if (mPendingRadioPowerOffAfterDataOff) {
692
693                 handlePowerOn();
694             }
695         }
696     }
697
698     protected void handlePowerOff() {
699         synchronized (this) {
700             if (!mPendingRadioPowerOffAfterDataOff) {
701
702                 powerOffRadioSafely(mDcTracker);
703             }
704         }
705     }
706
707     protected void powerOffRadioSafely() {
708         synchronized (this) {
709             if (!mPendingRadioPowerOffAfterDataOff) {
710
711                 handlePowerOff();
712             }
713         }
714     }
715
716     protected void handleDataOn() {
717         synchronized (this) {
718             if (mPendingRadioPowerOffAfterDataOff) {
719
720                 powerOnRadioSafely(mDcTracker);
721             }
722         }
723     }
724
725     protected void powerOnRadioSafely() {
726         synchronized (this) {
727             if (mPendingRadioPowerOffAfterDataOff) {
728
729                 handleDataOn();
730             }
731         }
732     }
733
734     protected void handlePowerOn() {
735         synchronized (this) {
736             if (mPendingRadioPowerOffAfterDataOff) {
737
738                 powerOnRadioSafely(mDcTracker);
739             }
740         }
741     }
742
743     protected void powerOnRadioSafely() {
744         synchronized (this) {
745             if (mPendingRadioPowerOffAfterDataOff) {
746
747                 handlePowerOn();
748             }
749         }
750     }
751
752     protected void handlePowerOff() {
753         synchronized (this) {
754             if (!mPendingRadioPowerOffAfterDataOff) {
755
756                 powerOffRadioSafely(mDcTracker);
757             }
758         }
759     }
760
761     protected void powerOffRadioSafely() {
762         synchronized (this) {
763             if (!mPendingRadioPowerOffAfterDataOff) {
764
765                 handlePowerOff();
766             }
767         }
768     }
769
770     protected void handleDataOff() {
771         synchronized (this) {
772             if (!mPendingRadioPowerOffAfterDataOff) {
773
774                 powerOffRadioSafely(mDcTracker);
775             }
776         }
777     }
778
779     protected void powerOffRadioSafely() {
780         synchronized (this) {
781             if (!mPendingRadioPowerOffAfterDataOff) {
782
783                 handleDataOff();
784             }
785         }
786     }
787
788     protected void handlePowerOn() {
789         synchronized (this) {
790             if (mPendingRadioPowerOffAfterDataOff) {
791
792                 powerOnRadioSafely(mDcTracker);
793             }
794         }
795     }
796
797     protected void powerOnRadioSafely() {
798         synchronized (this) {
799             if (mPendingRadioPowerOffAfterDataOff) {
800
801                 handlePowerOn();
802             }
803         }
804     }
805
806     protected void handlePowerOff() {
807         synchronized (this) {
808             if (!mPendingRadioPowerOffAfterDataOff) {
809
810                 powerOffRadioSafely(mDcTracker);
811             }
812         }
813     }
814
815     protected void powerOffRadioSafely() {
816         synchronized (this) {
817             if (!mPendingRadioPowerOffAfterDataOff) {
818
819                 handlePowerOff();
820             }
821         }
822     }
823
824     protected void handleDataOn() {
825         synchronized (this) {
826             if (mPendingRadioPowerOffAfterDataOff) {
827
828                 powerOnRadioSafely(mDcTracker);
829             }
830         }
831     }
832
833     protected void powerOnRadioSafely() {
834         synchronized (this) {
835             if (mPendingRadioPowerOffAfterDataOff) {
836
837                 handleDataOn();
838             }
839         }
840     }
841
842     protected void handlePowerOn() {
843         synchronized (this) {
844             if (mPendingRadioPowerOffAfterDataOff) {
845
846                 powerOnRadioSafely(mDcTracker);
847             }
848         }
849     }
850
851     protected void powerOnRadioSafely() {
852         synchronized (this) {
853             if (mPendingRadioPowerOffAfterDataOff) {
854
855                 handlePowerOn();
856             }
857         }
858     }
859
860     protected void handlePowerOff() {
861         synchronized (this) {
862             if (!mPendingRadioPowerOffAfterDataOff) {
863
864                 powerOffRadioSafely(mDcTracker);
865             }
866         }
867     }
868
869     protected void powerOffRadioSafely() {
870         synchronized (this) {
871             if (!mPendingRadioPowerOffAfterDataOff) {
872
873                 handlePowerOff();
874             }
875         }
876     }
877
878     protected void handleDataOn() {
879         synchronized (this) {
880             if (mPendingRadioPowerOffAfterDataOff) {
881
882                 powerOnRadioSafely(mDcTracker);
883             }
884         }
885     }
886
887     protected void powerOnRadioSafely() {
888         synchronized (this) {
889             if (mPendingRadioPowerOffAfterDataOff) {
890
891                 handleDataOn();
892             }
893         }
894     }
895
896     protected void handlePowerOn() {
897         synchronized (this) {
898             if (mPendingRadioPowerOffAfterDataOff) {
899
900                 powerOnRadioSafely(mDcTracker);
901             }
902         }
903     }
904
905     protected void powerOnRadioSafely() {
906         synchronized (this) {
907             if (mPendingRadioPowerOffAfterDataOff) {
908
909                 handlePowerOn();
910             }
911         }
912     }
913
914     protected void handlePowerOff() {
915         synchronized (this) {
916             if (!mPendingRadioPowerOffAfterDataOff) {
917
918                 powerOffRadioSafely(mDcTracker);
919             }
920         }
921     }
922
923     protected void powerOffRadioSafely() {
924         synchronized (this) {
925             if (!mPendingRadioPowerOffAfterDataOff) {
926
927                 handlePowerOff();
928             }
929         }
930     }
931
932     protected void handleDataOn() {
933         synchronized (this) {
934             if (mPendingRadioPowerOffAfterDataOff) {
935
936                 powerOnRadioSafely(mDcTracker);
937             }
938         }
939     }
940
941     protected void powerOnRadioSafely() {
942         synchronized (this) {
943             if (mPendingRadioPowerOffAfterDataOff) {
944
945                 handleDataOn();
946             }
947         }
948     }
949
950     protected void handlePowerOn() {
951         synchronized (this) {
952             if (mPendingRadioPowerOffAfterDataOff) {
953
954                 powerOnRadioSafely(mDcTracker);
955             }
956         }
957     }
958
959     protected void powerOnRadioSafely() {
960         synchronized (this) {
961             if (mPendingRadioPowerOffAfterDataOff) {
962
963                 handlePowerOn();
964             }
965         }
966     }
967
968     protected void handlePowerOff() {
969         synchronized (this) {
970             if (!mPendingRadioPowerOffAfterDataOff) {
971
972                 powerOffRadioSafely(mDcTracker);
973             }
974         }
975     }
976
977     protected void powerOffRadioSafely() {
978         synchronized (this) {
979             if (!mPendingRadioPowerOffAfterDataOff) {
980
981                 handlePowerOff();
982             }
983         }
984     }
985
986     protected void handleDataOn() {
987         synchronized (this) {
988             if (mPendingRadioPowerOffAfterDataOff) {
989
990                 powerOnRadioSafely(mDcTracker);
991             }
992         }
993     }
994
995     protected void powerOnRadioSafely() {
996         synchronized (this) {
997             if (mPendingRadioPowerOffAfterDataOff) {
998
999                 handleDataOn();
1000            }
1001        }
1002    }
1003
1004    protected void handlePowerOn() {
1005        synchronized (this) {
1006            if (mPendingRadioPowerOffAfterDataOff) {
1007
1008                powerOnRadioSafely(mDcTracker);
1009            }
1010        }
1011    }
1012
1013    protected void powerOnRadioSafely() {
1014        synchronized (this) {
1015            if (mPendingRadioPowerOffAfterDataOff) {
1016
1017                handlePowerOn();
1018            }
1019        }
1020    }
1021
1022    protected void handlePowerOff() {
1023        synchronized (this) {
1024            if (!mPendingRadioPowerOffAfterDataOff) {
1025
1026                powerOffRadioSafely(mDcTracker);
1027            }
1028        }
1029    }
1030
1031    protected void powerOffRadioSafely() {
1032        synchronized (this) {
1033            if (!mPendingRadioPowerOffAfterDataOff) {
1034
1035                handlePowerOff();
1036            }
1037        }
1038    }
1039
1040    protected void handleDataOn() {
1041        synchronized (this) {
1042            if (mPendingRadioPowerOffAfterDataOff) {
1043
1044                powerOnRadioSafely(mDcTracker);
1045            }
1046        }
1047    }
1048
1049    protected void powerOnRadioSafely() {
1050        synchronized (this) {
1051            if (mPendingRadioPowerOffAfterDataOff) {
1052
1053                handleDataOn();
1054            }
1055        }
1056    }
1057
1058    protected void handlePowerOn() {
1059        synchronized (this) {
1060            if (mPendingRadioPowerOffAfterDataOff) {
1061
1062                powerOnRadioSafely(mDcTracker);
1063            }
1064        }
1065    }
1066
1067    protected void powerOnRadioSafely() {
1068        synchronized (this) {
1069            if (mPendingRadioPowerOffAfterDataOff) {
1070
1071                handlePowerOn();
1072            }
1073        }
1074    }
1075
1076    protected void handlePowerOff() {
1077        synchronized (this) {
1078            if (!mPendingRadioPowerOffAfterDataOff) {
1079
1080                powerOffRadioSafely(mDcTracker);
1081            }
1082        }
1083    }
1084
1085    protected void powerOffRadioSafely() {
1086        synchronized (this) {
1087            if (!mPendingRadioPowerOffAfterDataOff) {
1088
1089                handlePowerOff();
1090            }
1091        }
1092    }
1093
1094    protected void handleDataOn() {
1095        synchronized (this) {
1096            if (mPendingRadioPowerOffAfterDataOff) {
1097
1098                powerOnRadioSafely(mDcTracker);
1099            }
1100        }
1101    }
1102
1103    protected void powerOnRadioSafely() {
1104        synchronized (this) {
1105            if (mPendingRadioPowerOffAfterDataOff) {
1106
1107                handleDataOn();
1108            }
1109        }
1110    }
1111
1112    protected void handlePowerOn() {
1113        synchronized (this) {
1114            if (mPendingRadioPowerOffAfterDataOff) {
1115
1116                powerOnRadioSafely(mDcTracker);
1117            }
1118        }
1119    }
1120
1121    protected void powerOnRadioSafely() {
1122        synchronized (this) {
1123            if (mPendingRadioPowerOffAfterDataOff) {
1124
1125                handlePowerOn();
1126            }
1127        }
1128    }
1129
1130    protected void handlePowerOff() {
1131        synchronized (this) {
1132            if (!mPendingRadioPowerOffAfterDataOff) {
1133
1134                powerOffRadioSafely(mDcTracker);
1135            }
1136        }
1137    }
1138
1139    protected void powerOffRadioSafely() {
1140        synchronized (this) {
1141            if (!mPendingRadioPowerOffAfterDataOff) {
1142
1143                handlePowerOff();
1144            }
1145        }
1146    }
1147
1148    protected void handleDataOn() {
1149        synchronized (this) {
1150            if (mPendingRadioPowerOffAfterDataOff) {
1151
1152                powerOnRadioSafely(mDcTracker);
1153            }
1154        }
1155    }
1156
1157    protected void powerOnRadioSafely() {
1158        synchronized (this) {
1159            if (mPendingRadioPowerOffAfterDataOff) {
1160
1161                handleDataOn();
1162            }
1163        }
1164    }
1165
1166    protected void handlePowerOn() {
1167        synchronized (this) {
1168            if (mPendingRadioPowerOffAfterDataOff) {
1169
1170                powerOnRadioSafely(mDcTracker);
1171            }
1172        }
1173    }
1174
1175    protected void powerOnRadioSafely() {
1176        synchronized (this) {
1177            if (mPendingRadioPowerOffAfterDataOff) {
1178
1179                handlePowerOn();
1180            }
1181        }
1182    }
1183
1184    protected void handlePowerOff() {
1185        synchronized (this) {
1186            if (!mPendingRadioPowerOffAfterDataOff) {
1187
1188                powerOffRadioSafely(mDcTracker);
1189            }
1190        }
1191    }
1192
1193    protected void powerOffRadioSafely() {
1194        synchronized (this) {
1195            if (!mPendingRadioPowerOffAfterDataOff) {
1196
1197                handlePowerOff();
1198            }
1199        }
1200    }
1201
1202    protected void handleDataOn() {
1203        synchronized (this) {
1204            if (mPendingRadioPowerOffAfterDataOff) {
1205
1206                powerOnRadioSafely(mDcTracker);
1207            }
1208        }
1209    }
1210
1211    protected void powerOnRadioSafely() {
1212        synchronized (this) {
1213            if (mPendingRadioPowerOffAfterDataOff) {
1214
1215                handleDataOn();
1216            }
1217        }
1218    }
1219
1220    protected void handlePowerOn() {
1221        synchronized (this) {
1222            if (mPendingRadioPowerOffAfterDataOff) {
1223
1224                powerOnRadioSafely(mDcTracker);
1225            }
1226        }
1227    }
1228
1229    protected void powerOnRadioSafely() {
1230        synchronized (this) {
1231            if (mPendingRadioPowerOffAfterDataOff) {
1232
1233                handlePowerOn();
1234            }
1235        }
1236    }
1237
1238    protected void handlePowerOff() {
1239        synchronized (this) {
1240            if (!mPendingRadioPowerOffAfterDataOff) {
1241
1242                powerOffRadioSafely(mDcTracker);
1243            }
1244        }
1245    }
1246
1247    protected void powerOffRadioSafely() {
1248        synchronized (this) {
1249            if (!mPendingRadioPowerOffAfterDataOff) {
1250
1251                handlePowerOff();
1252            }
1253        }
1254    }
1255
1256    protected void handleDataOn() {
1257        synchronized (this) {
1258            if (mPendingRadioPowerOffAfterDataOff) {
1259
1260                powerOnRadioSafely(mDcTracker);
1261            }
1262        }
1263    }
1264
1265    protected void powerOnRadioSafely() {
1266        synchronized (this) {
1267            if (mPendingRadioPowerOffAfterDataOff) {
1268
1269                handleDataOn();
1270            }
1271        }
1272    }
1273
1274    protected void handlePowerOn() {
1275        synchronized (this) {
1276            if (mPendingRadioPowerOffAfterDataOff) {
1277
1278                powerOnRadioSafely(mDcTracker);
1279            }
1280        }
1281    }
1282
1283    protected void powerOnRadioSafely() {
1284        synchronized (this) {
1285            if (mPendingRadioPowerOffAfterDataOff) {
1286
1287                handlePowerOn();
1288            }
1289        }
1290    }
1291
1292    protected void handlePowerOff() {
1293        synchronized (this) {
1294            if (!mPendingRadioPowerOffAfterDataOff) {
1295
1296                powerOffRadioSafely(mDcTracker);
1297            }
1298        }
1299    }
1300
1301    protected void powerOffRadioSafely() {
1302        synchronized (this) {
1303            if (!mPendingRadioPowerOffAfterDataOff) {
1304
1305                handlePowerOff();
1306            }
1307        }
1308    }
1309
1310    protected void handleDataOn() {
1311        synchronized (this) {
1312            if (mPendingRadioPowerOffAfterDataOff) {
1313
1314                powerOnRadioSafely(mDcTracker);
1315            }
1316        }
1317    }
1318
1319    protected void powerOnRadioSafely() {
1320        synchronized (this) {
1321            if (mPendingRadioPowerOffAfterDataOff) {
1322
1323                handleDataOn();
1324            }
1325        }
1326    }
1327
1328    protected void handlePowerOn() {
1329        synchronized (this) {
1330            if (mPendingRadioPowerOffAfterDataOff) {
1331
1332                powerOnRadioSafely(mDcTracker);
1333            }
1334        }
1335    }
1336
1337    protected void powerOnRadioSafely() {
1338        synchronized (this) {
1339            if (mPendingRadioPowerOffAfterDataOff) {
1340
1341                handlePowerOn();
1342            }
1343        }
1344    }
1345
1346    protected void handlePowerOff() {
1347        synchronized (this) {
1348            if (!mPendingRadioPowerOffAfterDataOff) {
1349
1350                powerOffRadioSafely(mDcTracker);
1351            }
1352        }
1353    }
1354
1355    protected void powerOffRadioSafely() {
1356        synchronized (this) {
1357            if (!mPendingRadioPowerOffAfterDataOff) {
1358
1359                handlePowerOff();
1360            }
1361        }
1362    }
1363
1364    protected void handleDataOn() {
1365        synchronized (this) {
1366            if (mPendingRadioPowerOffAfterDataOff) {
1367
1368                powerOnRadioSafely(mDcTracker);
1369            }
1370        }
1371    }
1372
1373    protected void powerOnRadioSafely() {
1374        synchronized (this) {
1375            if (mPendingRadioPowerOffAfterDataOff) {
1376
1377                handleDataOn();
1378            }
1379        }
1380    }
1381
1382    protected void handlePowerOn() {
1383        synchronized (this) {
1384            if (mPendingRadioPowerOffAfterDataOff) {
1385
1386                powerOnRadioSafely(mDcTracker);
1387            }
1388        }
1389    }
1390
1391    protected void powerOnRadioSafely() {
1392        synchronized (this) {
1393            if (mPendingRadioPowerOffAfterDataOff) {
1394
1395                handlePowerOn();
1396            }
1397        }
1398    }
1399
1400    protected void handlePowerOff() {
1401        synchronized (this) {
1402            if (!mPendingRadioPowerOffAfterDataOff) {
1403
1404                powerOffRadioSafely(mDcTracker);
1405            }
1406        }
1407    }
1408
1409    protected void powerOffRadioSafely() {
1410        synchronized (this) {
1411            if (!mPendingRadioPowerOffAfterDataOff) {
1412
1413                handlePowerOff();
1414            }
1415        }
1416    }
1417
1418    protected void handleDataOn() {
1419        synchronized (this) {
1420            if (mPendingRadioPowerOffAfterDataOff) {
1421
1422                powerOnRadioSafely(mDcTracker);
1423            }
1424        }
1425    }
1426
1427    protected void powerOnRadioSafely() {
1428        synchronized (this) {
1429            if (mPendingRadioPowerOffAfterDataOff) {
1430
1431                handleDataOn();
1432            }
1433        }
1434    }
1435
1436    protected void handlePowerOn() {
1437        synchronized (this) {
1438            if (mPendingRadioPowerOffAfterDataOff) {
1439
1440                powerOnRadioSafely(mDcTracker);
1441            }
1442        }
1443    }
1444
1445    protected void powerOnRadioSafely() {
1446        synchronized (this) {
1447            if (mPendingRadioPowerOffAfterDataOff) {
1448
1449                handlePowerOn();
1450            }
1451        }
1452    }
1453
1454    protected void handlePowerOff() {
1455        synchronized (this) {
1456            if (!mPendingRadioPowerOffAfterDataOff) {
1457
1458                powerOffRadioSafely(mDcTracker);
1459            }
1460        }
1461    }
1462
1463    protected void powerOffRadioSafely() {
1464        synchronized (this) {
1465            if (!mPendingRadioPowerOffAfterDataOff) {
1466
1467                handlePowerOff();
1468            }
1469        }
1470    }
1471
1472    protected void handleDataOn() {
1473        synchronized (this) {
1474            if (mPendingRadioPowerOffAfterDataOff) {
1475
1476                powerOnRadioSafely(mDcTracker);
1477            }
1478        }
1479    }
1480
1481    protected void powerOnRadioSafely() {
1482        synchronized (this) {
1483            if (mPendingRadioPowerOffAfterDataOff) {
1484
1485                handleDataOn();
1486            }
1487        }
1488    }
1489
1490    protected void handlePowerOn() {
1491        synchronized (this) {
1492            if (mPendingRadioPowerOffAfterDataOff) {
1493
1494                powerOnRadioSafely(mDcTracker);
1495            }
1496        }
1497    }
1498
1499    protected void powerOnRadioSafely() {
1500        synchronized (this) {
1501            if (mPendingRadioPowerOffAfterDataOff) {
1502
1503                handlePowerOn();
1504            }
1505        }
1506    }
1507
1508    protected void handlePowerOff() {
1509        synchronized (this) {
1510            if (!mPendingRadioPowerOffAfterDataOff) {
1511
1512                powerOffRadioSafely(mDcTracker);
1513            }
1514        }
1515    }
1516
1517    protected void powerOffRadioSafely() {
1518        synchronized (this) {
1519            if (!mPendingRadioPowerOffAfterDataOff) {
1520
1521                handlePowerOff();
1522            }
1523        }
1524    }
1525
1526    protected void handleDataOn() {
1527        synchronized (this) {
1528            if (mPendingRadioPowerOffAfterDataOff) {
1529
1530                powerOnRadioSafely(mDcTracker);
1531            }
1532        }
1533    }
1534
1535    protected void powerOnRadioSafely() {
1536        synchronized (this) {
1537            if (mPendingRadioPowerOffAfterDataOff) {
1538
1539                handleDataOn();
1540            }
1541        }
1542    }
1543
1544    protected void handlePowerOn() {
1545        synchronized (this) {
1546            if (mPendingRadioPowerOffAfterDataOff) {
1547
1548                powerOnRadioSafely(mDcTracker);
1549            }
1550        }
1551    }
1552
1553    protected void powerOnRadioSafely() {
1554        synchronized (this) {
1555            if (mPendingRadioPowerOffAfterDataOff) {
1556
1557                handlePowerOn();
1558            }
1559        }
1560    }
1561
1562    protected void handlePowerOff() {
1563        synchronized (this) {
1564            if (!mPendingRadioPowerOffAfterDataOff) {
1565
1566                powerOffRadioSafely(mDcTracker);
1567            }
1568        }
1569    }
1570
1571    protected void powerOffRadioSafely() {
1572        synchronized (this) {
1573            if (!mPendingRadioPowerOffAfterDataOff) {
1574
1575                handlePowerOff();
1576            }
1577        }
1578    }
1579
1580    protected void handleDataOn() {
1581        synchronized (this) {
1582            if (mPendingRadioPowerOffAfterDataOff) {
1583
1584                powerOnRadioSafely(mDcTracker);
1585            }
1586        }
1587    }
1588
1589    protected void powerOnRadioSafely() {
1590        synchronized (this) {
1591            if (mPendingRadioPowerOffAfterDataOff) {
1592
1593                handleDataOn();
1594            }
1595        }
1596    }
1597
1598    protected void handlePowerOn() {
1599        synchronized (this) {
1600            if (mPendingRadioPowerOffAfterDataOff) {
1601
1602                powerOnRadioSafely(mDcTracker);
1603            }
1604        }
1605    }
1606
1607    protected void powerOnRadioSafely() {
1608        synchronized (this) {
1609            if (mPendingRadioPowerOffAfterDataOff) {
1610
1611                handlePowerOn();
1612            }
1613        }
1614    }
1615
1616    protected void handlePowerOff() {
1617        synchronized (this) {
1618            if (!mPendingRadioPowerOffAfterDataOff) {
1619
1620                powerOffRadioSafely(mDcTracker);
1621            }
1622        }
1623    }
1624
1625    protected void powerOffRadioSafely() {
1626        synchronized (this) {
1627            if (!mPendingRadioPowerOffAfterDataOff) {
1628
1629                handlePowerOff();
1630            }
1631        }
1632    }
1633
1634    protected void handleDataOn() {
1635        synchronized (this) {
1636            if (mPendingRadioPowerOffAfterDataOff) {
1637
1638                powerOnRadioSafely(mDcTracker);
1639            }
1640        }
1641    }
1642
1643    protected void powerOnRadioSafely() {
1644        synchronized (this) {
1645            if (mPendingRadioPowerOffAfterDataOff) {
1646
1647                handleDataOn();
1648            }
1649        }
1650    }
1651
1652    protected void handlePowerOn() {
1653        synchronized (this) {
1654            if (mPendingRadioPowerOffAfterDataOff) {
1655
1656                powerOnRadioSafely(mDcTracker);
1657            }
1658        }
1659    }
1660
1661    protected void powerOnRadioSafely() {
1662        synchronized (this) {
1663            if (mPendingRadioPowerOffAfterDataOff) {
1664
1665                handlePowerOn();
1666            }
1667        }
1668    }
1669
1670    protected void handlePowerOff() {
1671        synchronized (this) {
1672            if (!mPendingRadioPowerOffAfterDataOff) {
1673
1674                powerOffRadioSafely(mDcTracker);
1675            }
1676        }
1677    }
1678
1679    protected void powerOffRadioSafely() {
1680        synchronized (this) {
1681            if (!mPendingRadioPowerOffAfterDataOff) {
1682
1683                handlePowerOff();
1684            }
1685        }
1686    }
1687
1688    protected void handleDataOn() {
1689        synchronized (this) {
1690            if (mPendingRadioPowerOffAfterDataOff) {
1691
1692                powerOnRadioSafely(mDcTracker);
1693            }
1694        }
1695    }
1696
1697    protected void powerOnRadioSafely() {
1698        synchronized (this) {
1699            if (mPendingRadioPowerOffAfterDataOff) {
1700
1701                handleDataOn();
1702            }
1703        }
1704    }
1705
1706    protected void handlePowerOn() {
1707        synchronized (this) {
1708            if (mPendingRadioPowerOffAfterDataOff) {
1709
1710                powerOnRadioSafely(mDcTracker);
1711            }
1712        }
1713    }
1714
1715    protected void powerOnRadioSafely() {
1716        synchronized (this) {
1717            if (mPendingRadioPowerOffAfterDataOff) {
1718
1719                handlePowerOn();
1720            }
1721        }
1722    }
1723
1724    protected void handlePowerOff() {
1725        synchronized (this) {
1726            if (!mPendingRadioPowerOffAfterDataOff) {
1727
1728                powerOffRadioSafely(mDcTracker);
1729            }
1730        }
1731    }
1732
1733    protected void powerOffRadioSafely() {
1734        synchronized (this) {
1735            if (!mPendingRadioPowerOffAfterDataOff) {
1736
1737                handlePowerOff();
1738            }
1739        }
1740    }
1741
1742    protected void handleDataOn() {
1743        synchronized (this) {
1744            if (mPendingRadioPowerOffAfterDataOff) {
1745
1746                powerOnRadioSafely(mDcTracker);
1747            }
1748        }
1749    }
1750
1751    protected void powerOnRadioSafely() {
1752        synchronized (this) {
1753            if (mPendingRadioPowerOffAfterDataOff) {
1754
1755                handleDataOn();
1756            }
1757        }
1758    }
1759
1760    protected void handlePowerOn() {
1761        synchronized (this) {
1762            if (mPendingRadioPowerOffAfterDataOff) {
1763
1764                powerOnRadioSafely(mDcTracker);
1765            }
1766        }
1767    }
1768
1769    protected void powerOnRadioSafely() {
1770        synchronized (this) {
1771            if (mPendingRadioPowerOffAfterDataOff) {
1772
1773                handlePowerOn();
1774            }
1775        }
1776    }
1777
1778    protected void handlePowerOff() {
1779        synchronized (this) {
1780            if (!mPendingRadioPowerOffAfterDataOff) {
1781
1782                powerOffRadioSafely(mDcTracker);
1783            }
1784        }
1785    }
1786
1787    protected void powerOffRadioSafely() {
1788        synchronized (this) {
1789            if (!mPendingRadioPowerOffAfterDataOff) {
1790
1791                handlePowerOff();
1792            }
1793        }
1794    }
1795
1796    protected void handleDataOn() {
1797        synchronized (this) {
1798            if (mPendingRadioPowerOffAfterDataOff) {
1799
1800                powerOnRadioSafely(mDcTracker);
1801            }
1802        }
1803    }
1804
1805    protected void powerOnRadioSafely() {
1806        synchronized (this) {
1807            if (mPendingRadioPowerOffAfterDataOff) {
1808
1809                handleDataOn();
1810            }
1811        }
1812    }
1813
1814    protected void handlePowerOn() {
1815        synchronized (this) {
1816            if (mPendingRadioPowerOffAfterDataOff) {
1817
1818                powerOnRadioSafely(mDcTracker);
1819            }
1820        }
1821    }
1822
1823    protected void powerOnRadioSafely() {
1824        synchronized (this) {
1825            if (mPendingRadioPowerOffAfterDataOff) {
1826
1827                handlePowerOn();
1828            }
1829        }
1830    }
1831
1832    protected void handlePowerOff() {
1833        synchronized (this) {
1834            if (!mPendingRadioPowerOffAfterDataOff) {
1835
1836                powerOffRadioSafely(mDcTracker);
1837            }
1838        }
1839    }
1840
1841    protected void powerOffRadioSafely() {
1842        synchronized (this) {
1843            if (!mPendingRadioPowerOffAfterDataOff) {
1844
1845                handlePowerOff();
1846            }
1847        }
1848    }
1849
1850    protected void handleDataOn() {
1851        synchronized (this) {
1852            if (mPendingRadioPowerOffAfterDataOff) {
1853
1854                powerOnRadioSafely(mDcTracker);
1855            }
1856        }
1857    }
1858
1859    protected void powerOnRadioSafely() {
1860        synchronized (this) {
1861            if (mPendingRadioPowerOffAfterDataOff) {
1862
1863                handleDataOn();
1864            }
1865        }
1866    }
1867
1868    protected void handlePowerOn() {
1869        synchronized (this) {
1870            if (mPendingRadioPowerOffAfterDataOff) {
1871
1872                powerOnRadioSafely(mDcTracker);
1873            }
1874        }
1875    }
1876
1877    protected void powerOnRadioSafely() {
1878        synchronized (this) {
1879            if (mPendingRadioPowerOffAfterDataOff) {
1880
1881                handlePowerOn();
1882            }
1883        }
1884    }
1885
1886    protected void handlePowerOff() {
1887        synchronized (this) {
1888            if (!mPendingRadioPowerOffAfterDataOff) {
1889
1890                powerOffRadioSafely(mDcTracker);
1891            }
1892        }
1893    }
1894
1895    protected void powerOffRadioSafely() {
1896        synchronized (this) {
1897            if (!mPendingRadioPowerOffAfterDataOff) {
1898
1899                handlePowerOff();
1900            }
1901        }
1902    }
1903
1904    protected void handleDataOn() {
1905        synchronized (this) {
1906            if (mPendingRadioPowerOffAfterDataOff) {
1907
1908                powerOnRadioSafely(mDcTracker);
1909            }
1910        }
1911    }
1912
1913    protected void powerOnRadioSafely() {
1914        synchronized (this) {
1915            if (mPendingRadioPowerOffAfterDataOff) {
1916
1917                handleDataOn();
1918            }
1919
```

```

48  /**
49   * This intent is broadcast by the system when carrier config changes.
50   */
51  public static final String
52      ACTION_CARRIER_CONFIG_CHANGED = "android.telephony.action.CARRIER_CONFIG_CHANGED";

```

**Fig. 21** Definition of ACTION\_CARRIER\_CONFIG\_CHANGED

unprotected intents periodically. We find that two intents can be exploited to achieve the same attacking effect, which are ACTION\_MANAGED\_PROFILE\_ADDED and ACTION\_MANAGED\_PROFILE\_REMOVED.

Android uses the same piece of code to process the two intents, which are in /packages/apps/Launcher3/src/com/android/launcher3/LauncherModel.java, as shown in Fig. 24. The component LauncherModel registers a broadcast receiver to receive the two intents (Line 1281 and 1282). If any one of them is received, the function forceReload (Line 1284) will be invoked, in which the launch activity will be reloaded. Note that the launch activity corresponds to the homescreen. Hence, the shortcuts will be hidden if the launch activity reloads in a fast rate, as shown in Fig. 25.

## 6.5 NFC state cheating

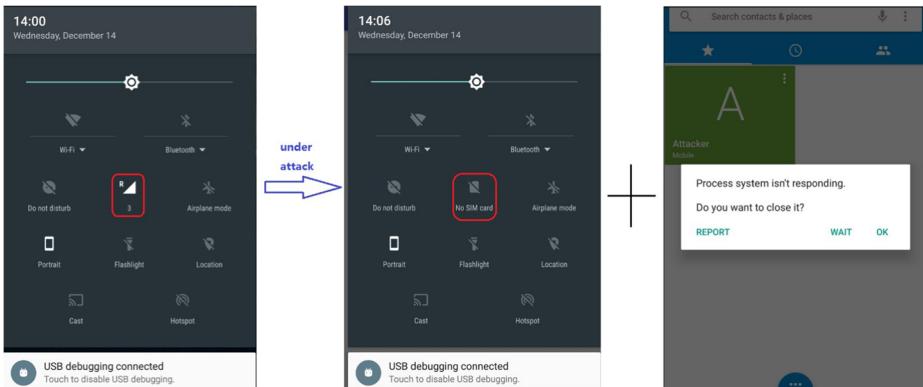
Near field communication (NFC) is a set of short-range wireless technologies, allowing users to share small payloads of data between an NFC tag and an Android-powered device, or between two Android-powered devices. Our attack can change the UI which presents the state of NFC, and thus users will be cheated. This attack takes advantage of the intent ACTION\_ADAPTER\_STATE\_CHANGED that is defined in

```

42 public class SimChangeReceiver extends BroadcastReceiver {
43     private final String TAG = "SimChangeReceiver";
44
45     @Override
46     public void onReceive(Context context, Intent intent) {
47
48         switch (action) {
49
50             case CarrierConfigManager.ACTION_CARRIER_CONFIG_CHANGED:
51
52                 if (!isUserSet) {
53                     // Preserve the previous setting for "isVisualVoicemailEnabled" if it is
54                     // set by the user, otherwise, set this value for the first time.
55                     VisualVoicemailSettingsUtil.setVisualVoicemailEnabled(context, phoneAccount,
56                         isEnabled, /*isUserSet */ false);
57                 }
58
59                 if (isEnabled) {
60                     LocalLogHelper.log(TAG, "Sim state or carrier config changed: requesting"
61                         + " activation for " + phoneAccount.getId());
62
63                     // Add a phone state listener so that changes to the communication channels
64                     // can be recorded.
65                     OmtppVmSourceManager.getInstance(context).addPhoneStateListener(
66                         phoneAccount);
67                     carrierConfigHelper.startActivation();
68
69                 } else {
70                     // It may be that the source was not registered to begin with but we want
71                     // to run through the steps to remove the source just in case.
72                     OmtppVmSourceManager.getInstance(context).removeSource(phoneAccount);
73                     Log.v(TAG, "Sim change for disabled account.");
74
75                 }
76             }
77         }
78     }
79 }

```

**Fig. 22** Vulnerable code exploited by SIM card removal



**Fig. 23** Symptom after SIM card removal attack

/frameworks/base/core/java/android/nfc/NfcAdapter.java, as shown in Fig. 26.

The component NfcEnabler processes the intent and updates UI, as shown in Fig. 27. In particular, the callback function of the registered broadcast receiver invokes handleNfcStateChanged (Line 49) to process the intent. In this function, Android updates UI according to the change of states. Please note that Android defines four states to maintain NFC component (i.e., STATE\_OFF, STATE\_TURNING\_ON, STATE\_ON, and STATE\_TURNING\_OFF), as shown in Fig. 26.

UI changes according to state transitions. To be specific, if NfcEnabler thinks the state is STATE\_OFF, the state of NFC slider (as shown in Fig. 28) will be unchecked and can be changed by tapping. If NfcEnabler considers the state to be STATE\_ON, the state of NFC slider will be checked and can be changed. If NfcEnabler considers the state to be STATE\_TURNING\_ON, the state of NFC slider will be checked, but it can not be changed by finger tapping. If NfcEnabler thinks the state is STATE\_TURNING\_OFF, the state of NFC slider will be unchecked and it can not be changed by finger tapping. Though source code inspection, we find the that both STATE\_TURNING\_ON and STATE\_TURNING\_OFF are intermediate states between STATE\_OFF and STATE\_ON. Therefore, if UI is in either intermediate states, the NFC slider can not respond to user interactions.

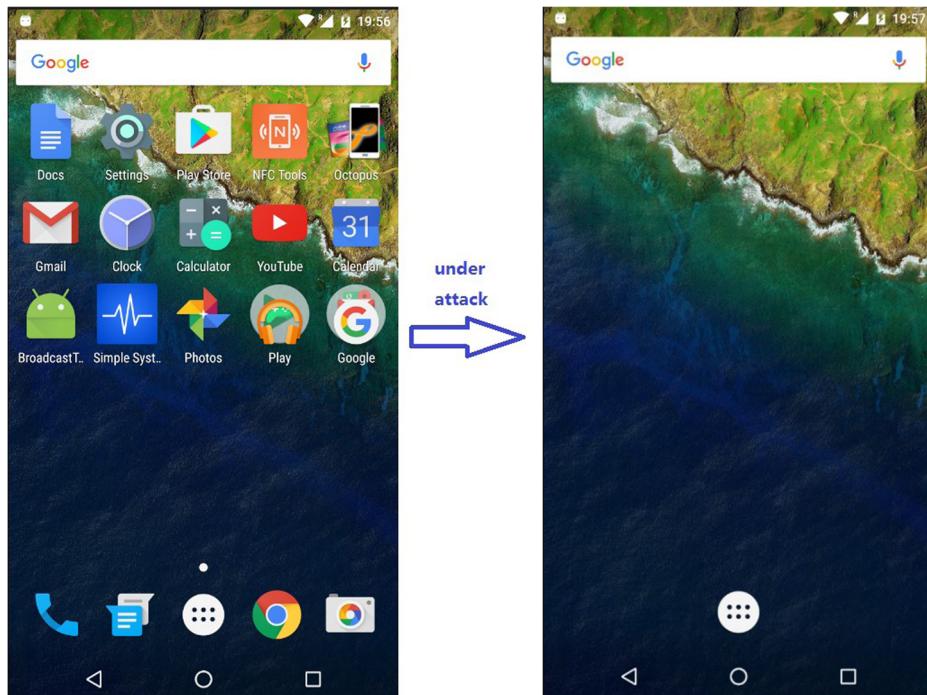
Our attack sends the unprotected intent that sets the attached data EXTRA\_ADAPTER\_STATE as STATE\_ON. As a consequence, the UI indicates that NFC is

```

1269     public void onReceive(Context context, Intent intent) {
...
1281         } else if (LauncherAppsCompat.ACTION_MANAGED_PROFILE_ADDED.equals(action)
1282             || LauncherAppsCompat.ACTION_MANAGED_PROFILE_REMOVED.equals(action)) {
1283             UserManagerCompat.getInstance(context).enableAndResetCache();
1284             forceReload();
1285         }
1286     }
1287
1288     void forceReload() {
1289         resetLoadedState(true, true);
1290
1291         // Do this here because if the launcher activity is running it will be restarted.
1292         // If it's not running startLoaderFromBackground will merely tell it that it needs
1293         // to reload.
1294         startLoaderFromBackground();
1295     }

```

**Fig. 24** Vulnerable code exploited by homescreen hiding



**Fig. 25** Symptom after homescreen hiding attack

enabled, however, the real state of NFC component is still turned off, as shown in Fig. 28. Our attack can also send the unprotected intent with EXTRA\_ADAPTER\_STATE setting as other values, making UI present other misleading states.

## 7 Threats to validity

### 7.1 Internal threats

There are some internal threats to the confidence in saying the study's results are correct. First, this work uses some programming patterns to find the definitions of intents, the declaration of protected intents. However, the patterns are concluded by manual code inspection. Hence, the number of intents, protected intents, and unprotected intents may not accurate

```

184     public static final String ACTION_ADAPTER_STATE_CHANGED =
185             "android.nfc.action.ADAPTER_STATE_CHANGED";
...
195     public static final String EXTRA_ADAPTER_STATE = "android.nfc.extra.ADAPTER_STATE";
196
197     public static final int STATE_OFF = 1;
198     public static final int STATE_TURNING_ON = 2;
199     public static final int STATE_ON = 3;
200     public static final int STATE_TURNING_OFF = 4;

```

**Fig. 26** Definition of ACTION\_ADAPTER\_STATE\_CHANGED and several states

```

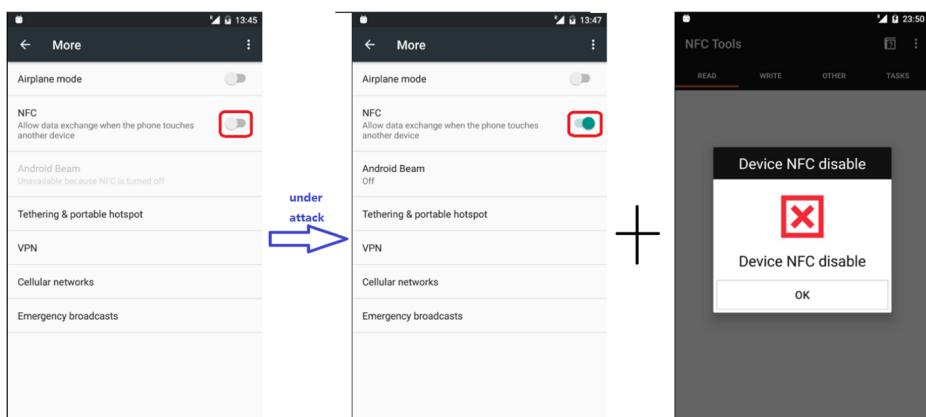
44     private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
45         @Override
46         public void onReceive(Context context, Intent intent) {
47             String action = intent.getAction();
48             if (NfcAdapter.ACTION_ADAPTER_STATE_CHANGED.equals(action)) {
49                 handleNfcStateChanged(intent.getIntExtra(NfcAdapter.EXTRA_ADAPTER_STATE,
50                                         NfcAdapter.STATE_OFF));
51             }
52         }
53     };
54
55     private void handleNfcStateChanged(int newState) {
56         switch (newState) {
57             case NfcAdapter.STATE_OFF:
58                 mSwitch.setChecked(false);
59                 mSwitch.setEnabled(true);
60                 mAndroidBeam.setEnabled(false);
61                 mAndroidBeam.setSummary(R.string.android_beam_disabled_summary);
62                 break;
63             case NfcAdapter.STATE_ON:
64                 mSwitch.setChecked(true);
65                 mSwitch.setEnabled(true);
66                 mAndroidBeam.setEnabled(!mBeamDisallow);
67                 if (mNfcAdapter.isNdefPushEnabled() && !mBeamDisallow) {
68                     mAndroidBeam.setSummary(R.string.android_beam_on_summary);
69                 } else {
70                     mAndroidBeam.setSummary(R.string.android_beam_off_summary);
71                 }
72                 break;
73             case NfcAdapter.STATE_TURNING_ON:
74                 mSwitch.setChecked(true);
75                 mSwitch.setEnabled(false);
76                 mAndroidBeam.setEnabled(false);
77                 break;
78             case NfcAdapter.STATE_TURNING_OFF:
79                 mSwitch.setChecked(false);
80                 mSwitch.setEnabled(false);
81                 mAndroidBeam.setEnabled(false);
82                 break;
83         }
84     }
85 }
86
87

```

**Fig. 27** Vulnerable code exploited by NFC state cheating

since developers can declare intents while not following the patterns. Moreover, we classify unprotected intents according to their names. However, one intent may be processed by different components, increasing the difficulty of accurate classification.

Additionally, the explanations of the vulnerabilities presented in Section 6 depend on manual code analysis, that may be inaccurate. The reason lies that after the observation of attacks, we check the source manually, which is not guaranteed to be accurate. In the future, we will validate the causes of attacks through dynamic debugging. Besides, the experiments of ATUIN randomly select 20 intents. However, the number and selection of intents can



**Fig. 28** Symptom after NFC state cheating attack

influence the experimental results. For example, the process of an intent involving heavy I/O operations may cause higher CPU utilization ratio than the intent whose processing code is much simpler. We leave the investigation of the selection strategy as future work.

## 7.2 External threats

There are several external threats to the confidence in stating whether the study's results are applicable to other groups. First, we conduct all experiments on one device, Huawei Nexus 6P. We plan to carry out similar studies on other Android devices in future. Second, this work only studies Android 6, leaving the investigation of other versions as future work. Third, this study uses fixed programming patterns to find the definitions of intents and the declarations of protected intents. Other versions of Android may not use the same patterns.

## 8 Related work

This work relates to the following research topics on Android, which are asynchronous-programming-related bugs, DoS attacks, resources-depletion attacks, intents-related bugs, and other performance bugs. This section briefly discusses the five categories of related studies separately.

### 8.1 Asynchronous-programming-related bugs

The heavy workload in main thread is a well-known cause of many performance problems (Liu et al. 2014). Android provides several async constructs (e.g., AsyncTask, IntentService and AsyncTaskLoader) that enable developers to put long-running tasks into background threads. However, existing studies (Lin et al. 2014, 2015; Chen et al. 2016; Kang et al. 2016) show that developers have to use AsyncTask carefully to avoid security vulnerabilities.

ASYNCHRONIZER (Lin et al. 2014) is an automated refactoring tool that enables developers to extract long-running operations into AsyncTask and uses a points-to static analysis to determine the safety of the transformation. ASYNCDROID (Lin et al. 2015) is a refactoring tool which enables Android developers to transform existing improperly-used async constructs (i.e., AsyncTask) into correct constructs (i.e., IntentService).

DIAGDROID (Kang et al. 2016) is a UI performance diagnosis tool, which is able to profile the asynchronous executions in a task granularity, equipping it with low-overhead and high compatibility merits. Chen et al. (2016) is our previous work which investigates a new feature, Doze mode in Android 6. Chen et al. (2016) finds one unprotected intent in the code for implementing Doze mode and proposes several approaches to deplete battery power by exploiting the intent.

### 8.2 DoS attacks

This work discovers tens of vulnerabilities. By exploiting them, hackers can deny some critical services of Android 6, such as Wi-Fi, telephone signal, SIM card, and launch activities. As far as we know, none of the proposed attacks were covered by related studies. Huang et al. (2015) proposed a new type of vulnerabilities, Android stroke vulnerabilities (ASV) which can lead to system Services freezing and system server shutdown. ASV corresponds to a flaw in the design of the coarse-grained concurrency control in the core of Android,

**System Server**, leading to a chance of DoS attacks. Based on the vulnerability, hackers can launch attacks in a straightforward way: writing a simple loop to call normal Android APIs to easily craft several exploits.

Different with their previous work (Huang et al. 2015), Liu's group discovers another vulnerability in **System Server**, that is the flaw in designing of synchronous callback mechanism (Wang et al. 2016). By exploiting the vulnerability, they enable a malicious application to freeze critical system functionalities or soft-reboot the system immediately. After elaborate construction, they successfully launch other meaningful attacks, such as anti anti-virus, anti process-killer, hindering app updates or system patching.

Armando et al. propose a DoS attack that makes devices become totally unresponsive (Armando et al. 2012). Their work bases on the observation that Android sets security policies to protect **Zygote**, a process enables fast start-up for new processes from being exploited by attacks. However, the protection is weak that can be bypassed easily, resulting in a large number of dummy processes until all memory resources are exhausted. Eian and Mjolsnes (Eian and Mjolsnes 2012) use formal method to identify deadlock vulnerability that causes DoS attacks in IEEE 802.11w protocol.

### 8.3 Resources-depletion attacks

This study implements ATUIN to attack CPU, leading to a high CPU utilization ratio. The related study aforementioned (Armando et al. 2012) can exhaust all memory resources. This section mainly focuses battery-draining attacks, which should be a severe threat to mobile devices since they are power-limited and not always plugged. Our previous work (Chen et al. 2016) drain battery silently by exploiting an unprotected intent.

Fiore et al. proposed to drain battery stealthily by sending the victim's browser with unhearable audio files (Fiore et al. 2014), for example, sounds below 20 Hz. As a result, the power is wasted by playing unhearable music. Researchers found that Android applications can deplete battery (deliberately or unintentionally) by misusing power management APIs. To reduce battery consumption aggressively, Android exports wakelock-related APIs to application programmers. Hence, applications can keep the smartphone awake by acquiring a wakelock, and then allow it to sleep after releasing the wakelock. However, programmers sometimes forget to release wakelocks in each path (Jindal et al. 2013a; Pathak et al. 2012), or place wakelocks in wrong places (Alam et al. 2014; Jindal et al. 2013b), incurring power waste or faulty program logic. NANSA (Bauer et al. 2015) holds a partial wakelock, preventing CPU going to sleep and then stimulates benign applications to do power-intensive work when the screen is off.

### 8.4 Intents-related bugs

INTENTFUZZER (Yang et al. 2014) generates intents to discover capability leaks of Android applications that finds more than 100 applications in Google play has at least one permission leak. Android system can be attacked by the web browsers which support intent scheme URLs. When parsing an intent scheme URL, the web browser will generate intents to launch activities. By exploiting intent scheme URLs, hackers can launch attacks including cookie file theft and universal XSS (Terada 2014, <http://www.mbsd.jp/Whitepaper/IntentScheme.pdf>). Schartner and Bürger (2012) propose to insert malicious processing functions into Android system to handle intents as hackers' will. Based on the idea, they successfully hack the applications secured by mTANs, such as web-banking.

## 8.5 Other performance bugs

Guo et al. (2013) developed a static analysis tool, Relda which detects energy and memory leaks as well as the resources never being released. Liu et al. (2014) analyze 70 real performance bugs from 8 Android applications and conclude three categories of performance bugs (i.e., GUI lagging, memory bloat, energy leak). Additionally, the authors propose PerfChecker, a static program analysis tool to identify two types of performance bugs: lengthy operations in the UI thread and violations of the view holder pattern. Linares-Vásquez et al. propose a taxonomy of practices and tools for detecting and fixing performance bottlenecks based on the survey with 485 developers (Linares-Vásquez et al. 2015). Xu et al. (2012); Zhang et al. (2012); Liu et al. (2014) leverages cost-benefit analysis to detect whether an Android application uses sensory data in a cost-ineffective way.

STRICTMODE (<http://developer.android.com/reference/android/os/StrictMode.html>) is a developer tool provided by Android that aims at finding blocking operations in main thread. To reduce application latency, TANGO (Gordon et al. 2015) offloads some workload from the smartphone to a remote server. TANGO replicates the application and executes it on both the client and the server, and allows either replica to lead the execution. Similar with TANGO, OUTATIME (Lee et al. 2015) performs game execution and rendering on remote servers on behalf of thin clients that simply send input and display output frames. SMARTIO (Nguyen et al. 2015) reduces the application delay by prioritizing reads over writes, and grouping them based on assigned priorities.

## In summary, our work differs from related studies in the following aspects

- We focus on the async construct, IntentService.
- We reveal that a lot of intents are unprotected from being manipulated by third-party applications.
- We discover tens of critical vulnerabilities which have not been reported before. To the best of our knowledge, our work is the *first* systematic study about hacking Android system by exploiting IntentService.

## 9 Conclusion

To reduce application latency, Android provides asynchronous programming which enables developers to put long-running tasks into background threads. This paper focuses on one async construct, IntentService. Through static program analysis, our work finds nearly one thousand unprotected intents which can be sent by third-party applications. Moreover, we implement a tool which is able to attack a CPU by exploiting the unprotected intents automatically. Furthermore, we discover tens of critical vulnerabilities that have not been reported before.

We plan to extend our work from the following aspects. First, we are interested in designing an automated approach to discover critical vulnerabilities like those in Section 6. Second, we plan to conduct a similar systematic study on other Android versions, such as Android 7 (the newest version), Android 5 which still leads the market share at 32% (Bandla <http://www.gadgetdetail.com/android-version-market-share-distribution/>). Moreover, we plan to test ATUIN in different settings (e.g., different numbers of intents, selecting different intents).

**Acknowledgements** This work is supported in part by the Hong Kong GRF (PolyU 152279/16E), the HKPolyU Research Grants (G-YBJX), Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892), the National Natural Science Foundation of China (Nos. 61402080, 61572115, 61502086, and 61572109), and China Postdoctoral Science Foundation founded project (No. 2014M562307). We specially thank Dr. Yajuan Tang from College of Engineering, Shantou University for her assist in improving our paper.

## References

- Alam, F., Panda, P.R., Tripathi, N., Sharma, N., & Narayan, S. (2014). Energy optimization in Android applications through wakelock placement. In *Proceedings of DATE* (pp. 1–4).
- Armando, A., Merlo, A., Migliardi, M., & Verderame, L. (2012). Would you mind forking this process? A denial of service attack on Android (and some countermeasures). In *Proceedings of IFIP SEC* (pp. 13–24).
- Bandla. Android Version Share: Lollipop still leads with 34%, Nougat at 0.4%. <http://www.gadgetdetail.com/android-version-market-share-distribution/>.
- Bauer, M., Coatsworth, M., & Moeller, J. (2015). NANSA: A no-attribution nosleep battery exhaustion attack for portable computing devices.
- Chen, T., Tang, H., Zhou, K., Zhang, X., & Lin, X. (2016). Silent Battery Draining Attack Against Android Systems by Subverting Doze Mode. In *Proceedings of the GlobeCom*.
- Eian, M., & Mjolsnes, S. (2012). A formal analysis of IEEE 802.11w deadlock vulnerabilities. In *Proceedings of INFOCOM*.
- Fiore, U., Palmieri, F., Castiglione, A., Loia, V., & De Santis, A. (2014). Multimedia-based battery drain attacks for android devices. In *Proceedings of CCNC* (pp. 145–150).
- Gordon, M.S., Hong, D.K., Chen, P.M., Flinn, J., Mahlke, S., & Mao, Z.M. (2015). Accelerating mobile applications through flip-flop replication. In *Proceedings of MobiSys* (pp. 137–150).
- Guo, C., Zhang, J., Yan, J., Zhang, Z., & Zhang, Y. (2013). Characterizing and detecting resource leaks in android applications. In *Proceedings of ASE* (pp. 389–398).
- Huang, H., Zhu, S., Chen, K., & Liu, P. (2015). From system services freezing to system server shutdown in android All you need is a loop in an app. In *Proceedings of CCS* (pp. 1236–1247).
- Jindal, A., Pathak, A., Hu, Y.C., & Midkiff, S. (2013a). Hypnos: understanding and treating sleep conflicts in smartphones. In *Proceedings of EuroSys* (pp. 253–266).
- Jindal, A., Pathak, A., Hu, Y.C., & Midkiff, S. (2013b). On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of HotPower* (pp. 1–5).
- Kang, Y., Zhou, Y., Xu, H., & Lyu, M.R. (2016). DiagDroid: Android performance diagnosis via anatomizing asynchronous executions. In *Proceedings of the FSE* (pp. 410–421).
- Lee, K., Chu, D., Cuervo, E., Kopf, J., Degtyarev, Y., Grizan, S., Wolman, A., & Flinn, J. (2015). Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of MobiSys* (pp. 151–165).
- Lin, Y., Radoi, C., & Dig, D. (2014). Retrofitting concurrency for android applications through refactoring. In *Proceedings of the FSE, 2014* (pp. 341–352).
- Lin, Y., Radoi, C., & Dig, D. (2015). Study and refactoring of android asynchronous programming. In *Proceedings of the ASE* (pp. 224–235). 2015.
- Linares-Vásquez, M., Vendome, C., Luo, Q., & Poshyvanyk, D. (2015). How developers detect and fix performance bottlenecks in android apps. In *Proceedings of ICSME* (pp. 352–361).
- Liu, Y., Xu, C., & Cheung, S.-C. (2014). Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of ICSE* (pp. 1013–1024).
- Nguyen, D.T., Zhou, G., Xing, G., Qi, X., Hao, Z., Peng, G., & Yang, Q. (2015). Reducing smartphone application delay through read/write isolation. In *Proceedings of Mobicys* (pp. 287–300).
- Pathak, A., Jindal, A., Hu, Y.C., & Midkiff, S.P. (2012). What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of MobiSys* (pp. 267–280).
- Schartner, P., & Bürger, S. (2012). Attacking Android's Intent Processing and First Steps towards Protecting it. Technical Report TR-syssec-12-01, Universität Klagenfurt.

- 
- Terada, T. (2014). Attacking Android browsers via intent scheme. [http://www.mbsd.jp/Whitepaper/Intent\\_Scheme.pdf](http://www.mbsd.jp/Whitepaper/Intent_Scheme.pdf).
- Wang, K., Zhang, Y., & Liu, P. (2016). Call me Back!: attacks on system server and system apps in android through synchronous callback. In *Proceedings of CCS* (pp. 92–103).
- Xu, G., Mitchell, N., Arnold, M., Rountev, A., Schonberg, E., & Sevitsky, G. (2012). Finding low-utility data structures. In *Proceedings of PLDI* (pp. 174–186).
- Yang, S., Yan, D., & Rountev, A. (2013). Testing for poor responsiveness in Android applications. In *Proceedings of the MOBS* (pp. 1–6).
- Yang, K., Zhuge, J., Wang, Y., Zhou, L., & Duan, H. (2014). Intentfuzzer: detecting capability leaks of android applications. In *Proceedings of ASIACCS* (pp. 531–536).
- Zhang, L., Gordon, M.S., Dick, R.P., Mao, Z., Dinda, P.A., & Yang, L. (2012). ADEL: An automated detector of energy leaks for smartphone applications. In *Proceedings of CODES+ISSS* (pp. 363–372).



**Ting Chen** received his Bachelor's degree in Applied Mathematics from University of Electronic Science and Technology of China (UESTC), 2007, Master's degree in Computer Application Technologies from UESTC, 2010, and Doctor's degree in Computer Software and Theory from UESTC, 2013. Now he is an Associate Professor in Cybersecurity Research Center, UESTC. His research domain includes software security, system security, and mobile security.



**Xiaoqi Li** received his Bachelor's degree from Central South University and his Master's degree from Chinese Academy of Sciences. Now he is a Ph.D. Student at The Hong Kong Polytechnic University. His research focuses on software security and privacy.



**Xiapu Luo** is a research assistant professor in the Department of Computing at The Hong Kong Polytechnic University. His research focuses on smartphone security, network security, and privacy and Internet measurement.



**Xiaosong Zhang** is a full professor in Cybersecurity Research Center, UESTC. His research focuses on network security, system security and data security.