BaseMirror: Automatic Reverse Engineering of Baseband Commands from Android's Radio Interface Layer

Wenqiang Li*
li.14488@osu.edu
The Ohio State University
Columbus, Ohio, USA

Haohuang Wen*
wen.423@osu.edu
The Ohio State University
Columbus, Ohio, USA

Zhiqiang Lin
zlin@cse.ohio-state.edu
The Ohio State University
Columbus, Ohio, USA

Abstract

In modern mobile devices, baseband is an integral component running on top of cellular processors to handle crucial radio communications. However, recent research reveals significant vulnerabilities in these basebands, posing serious security risks like remote code execution. Yet, effectively scrutinizing basebands remains a daunting task, as they run closed-source and proprietary software on vendor-specific chipsets. Existing analysis methods are limited by their dependence on manual processes and heuristic approaches, reducing their scalability. This paper introduces a novel approach to unveil security issues in basebands from a unique perspective: to uncover vendor-specific baseband commands from the *Radio Interface Layer (RIL)*, a hardware abstraction layer interfacing with basebands. To demonstrate this concept, we have designed and developed BaseMirror, a static binary analysis tool to automatically reverse engineer baseband commands from vendor-specific RIL binaries. It utilizes a bidirectional taint analysis algorithm to adeptly identify baseband commands from an enhanced control flow graph enriched with reconstructed virtual function calls. Our methodology has been applied to 28 vendor RIL libraries, encompassing a wide range of Samsung Exynos smartphone models on the market. Remarkably, BaseMirror has uncovered 873 unique baseband commands undisclosed to the public. Based on these results, we develop an automated attack discovery framework to successfully derive and validate 8 zero-day vulnerabilities that trigger denial of cellular service and arbitrary file access on a Samsung Galaxy A53 device. These findings have been reported and confirmed by Samsung and a bug bounty was awarded to us.

CCS Concepts

• **Security and privacy → Mobile and wireless security**.

Keywords

Baseband, Android Radio Interface Layer, Reverse Engineering

*Both authors contributed equally to this research.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690254

ACM Reference Format:
Wenqiang Li, Haohuang Wen, and Zhiqiang Lin. 2024. BaseMirror: Automatic Reverse Engineering of Baseband Commands from Android's Radio Interface Layer. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690254

1 Introduction

Baseband is a proprietary and mandatory component in mobile devices, responsible for overseeing all radio functions like voice calls, text messages (SMS), and cellular data connections. Essentially, the baseband serves as the primary interface enabling a smartphone to communicate with the external cellular network, specifically the base station towers. Consequently, the baseband firmware code is complicated, dealing with low-level signals and cellular states, which are transparent to end-users. To date, key manufacturers like Qualcomm, Samsung, and MediaTek have created closed-source software and integrated system-on-chips (SoCs) that are used by manufacturers such as Google and Apple to produce smartphones, tablets, and other cellular devices.

However, it is less known that the baseband operates as a self-contained and highly independent system with numerous undisclosed functionalities. Specifically, the baseband firmware runs on a dedicated *Cellular Processor (CP)* with a real-time operating system (RTOS), rendering it entirely separated from the mobile device's main processor, commonly referred to as the *Application Processor (AP)* [31]. While previous studies have identified various vendor-specific AT commands in smartphone firmware triggering unpublicized baseband functions [29, 55], these AT commands are typically used in legacy Android phone models. In fact, we notice that most recent Android devices use proprietary inter-process communication (IPC) mechanisms to communicate with the baseband [9, 33]. These baseband interfaces also have raised significant security concerns. For example, Samsung Galaxy devices' baseband was reported to contain remote file access (RFS) backdoors, providing unauthorized access to read, write, or delete files on the smartphone's local file system [28].

Even more concerning, recent demonstrations have revealed that mobile basebands are susceptible to exploitation, allowing the triggering of vulnerabilities like remote code execution (RCE) over the air [20], even extending to the latest 5G mobile devices [19]. Over the years, it has become evident that the baseband often suffers from inadequate security engineering, showcasing memory corruption vulnerabilities [18, 23, 40, 67] and non-compliance with cellular specifications [26, 31]. Simultaneously, high-privileged malware on a device can inject malicious requests into the baseband, activating security-sensitive functions. These exploitation pathways pose a serious risk, allowing malicious actors to compromise the device and jeopardize the user's security and privacy.

Current approaches involve scrutinizing baseband firmware implementations through reverse engineering (RE) to identify and address undesired behaviors and vulnerabilities in basebands. As of now, only a few tools have been developed for baseband RE, encompassing both static analysis [31, 38] and dynamic analysis [26], with some utilizing emulation methods [23, 40]. However, these tools come with several limitations, often relying on manual analysis and heuristics, and thus do not generalize well [31, 40]. Indeed, baseband RE is an exceedingly difficult task due to various factors such as undisclosed implementation details, diverse architectures, and the absence of debugging symbols.

This paper presents a novel approach, BaseMirror, which unveils baseband security issues from a new perspective. Instead of directly analyzing the baseband firmware, BaseMirror reverse-engineers vendor-specific baseband commands from the *Radio Interface Layer* (RIL) [2], which effectively serves as a *mirror* of the baseband interfaces. This "mirror-based" perspective has been successfully applied in previous research to uncover security vulnerabilities in automotive and IoT devices [58, 60, 62, 64]. In this work, our key insight is rooted in the generic architecture of Android mobile devices, where all communication between the Application Processor (AP) and Cellular Processor (CP) is processed and mediated by the RIL—a hardware abstraction layer between the radio hardware and the operating system. The critical commands implemented in the RIL thus expose exploitable attack surfaces targeting the AP and CP. As the RIL logic is typically integrated into a standalone shared library by the baseband hardware vendor (e.g., Samsung and Qualcomm), we consequently formulate it as a reverse engineering task to uncover vendor-specific baseband commands from the RIL binary code.

Nonetheless, developing such a static analysis pipeline poses three technical challenges. First, it requires a generic algorithm to comprehensively identify desired program paths generating these baseband commands so that they can be applied to various baseband versions and models without human intervention. Second, the inter-procedural call graph of the RIL binary needs to be recovered but it is challenging due to the massive use of virtual function calls [5, 14] that could not be resolved by off-the-shelf reverse engineering tools by default [42, 48, 54]. Third, it needs to accurately filter out commands not dedicated to the baseband, as the RIL binaries are still relatively complex, typically involving over 15K functions, and not all contribute to baseband communication.

We have addressed the above challenges and implemented a prototype of BaseMirror as an automatic static binary code analysis tool to reveal vendor-proprietary baseband commands from the Android RIL. It employs a bidirectional taint analysis algorithm to comprehensively track the data flow from fundamental system APIs invoked for baseband interactions. To facilitate this analysis, BaseMirror recovers the indirect function calls (especially virtual calls) by employing a lightweight and efficient algorithm to resolve the virtual call target and function. Subsequently, it discards commands not intended for the baseband, based on the system path associated with the corresponding command channels.

To evaluate BaseMirror, we have tested it with 28 vendor RIL libraries extracted from real Android firmware images, covering a wide range of Samsung smartphone models with the proprietary Exynos baseband. BaseMirror successfully extracted 873 unique vendor-specific commands. We further developed a dynamic validation framework on a real device to verify the correctness of 179 commands that do not involve external parameters, which shows no false positive. To assess the security implications of our results, we developed an automatic attack payload discovery tool that derives 8 new exploitable commands on a Samsung Galaxy A53. We demonstrate that these attack commands can be leveraged to trigger service disruption and denial-of-service attacks on the CP, including three new attacks that require re-flashing the baseband to recover. We also discover a new vulnerability enabling arbitrary access to the AP's file system. This vulnerability has been patched in the latest RIL firmware and a bug bounty was awarded.

In summary, we make the following contributions:

- We present a new approach to unveil security issues in basebands from the Radio Interface Layer, by reverse engineering vendor-specific baseband commands.
- We developed BaseMirror, a static binary analysis tool, and used that to extract 873 unique baseband commands from 28 different Samsung Exynos smartphones' firmware.
- We identified 8 zero-day vulnerabilities that exploit baseband commands to trigger denial-of-service attacks on the CP and arbitrary file accesses on the AP.
- We have released a proof-of-concept implementation of BaseMirror at https://github.com/OSUSecLab/BaseMirror.

2 Background and Motivation

Radio Interface Layer. In modern wireless systems, the Radio Interface Layer (RIL), a crucial element of the Android architecture [2], serves as the bridge between software and radio hardware. Initially specific to Android [2], RIL now broadly represents similar concepts in the hardware abstraction layer (HAL) [32]. As a vital interface, RIL enables seamless communication between the operating system and the radio modem, or baseband, essential for data transmission and reception over wireless networks. By abstracting radio hardware complexities, RIL provides a standardized interface that allows higher-level software, especially telephony applications, to interact with the radio module effectively. Its importance extends beyond basic functionality, enhancing interoperability, streamlining communication protocols, and optimizing resource use in mobile devices. For a comprehensive understanding, we outline the RIL architecture within a typical Android system in Figure 1 and elaborate on the primary RIL components subsequently.

RILD. The RIL Daemon (RILD) serves as the intermediary between the higher-level Android framework layer and the remaining RIL components [2]. More specifically, the RILD initializes the vendor RIL and LibRIL during system start-up. The initialization will bridge the communication between vendor RIL and LibRIL as the RILD will register the vendor RIL's communication interfaces (e.g., a `RIL_RadioFunctions` structure) to the LibRIL so that they can be invoked in the future. After the initialization, the RILD processes all communication events from the Android telephony service (via the RIL Java, or RILJ layer through Android's Inter-process communication mechanisms [3]) and further dispatches requests.

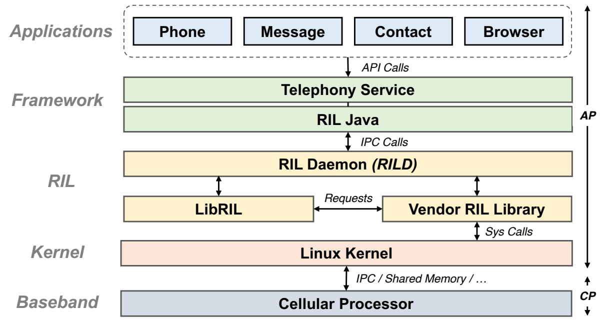


Figure 1: RIL architecture within an Android mobile device.

Direct Call
Virtual Call

```
1  IpcProtocol41::IpcTxGetCallList(this)
2  {  ...
3     // Construct Solicited Command to baseband
4     local_30 = 7;
5     local_2c = 0x602;
6     local_2a = 2;
7     // Virtual Call to IpcModem::SendMessage()
8     ((this+0x10)+0x30))(this+0x10, &local_30, 7);
9  }
10 IpcModem::SendMessage(this, char *param1, int param2)
11  {  ...
12    uVar1 = DoIoChannelRoutingTx(this, param1, param2);
13    ...
14 }
15 IpcModem::DoIoChannelRoutingTx(this, char *param1, int param2)
16 {  ...
17    // Virtual Call to IoChannel::Write()
18    uVar2 = ((this+0x30)+0x30))(this+0x30, param1, param2);
19    ...
20 }
21 IoChannel::Write(this, char *param1, int param2)
22 {  ...
23    // Write solicited command to baseband
24    sVar4 = write(*(this+8), param1, param2);
25    ...
26 }
```

Figure 2: A simplified running example of a solicited command to query call list from the baseband.

LibRIL. The LibRIL is a generic Android shared library (`.so`) that interacts with the vendor RIL through standard interfaces defined in the Android RIL source code [2]. To monitor the events and responses, LibRIL maintains an event loop thread that constantly listens to a series of file descriptors of interest. Once the LibRIL is notified of events from either the RILJ or the baseband, it triggers the corresponding handler functions to respond to the events. For instance, LibRIL will invoke the vendor RIL through its `onRequest` interface after receiving events from the upper framework layer. The Android Open Source Project (AOSP) [4] has defined a set of standard RIL commands to ensure basic cellular functionality, such as dialing and SMS.

Vendor RIL and the Reverse Engineering Opportunities. The Android platform, while maintaining generic components like RILD and LibRIL for standard RIL communication, also enables hardware vendors to craft their customized logic and protocols within a shared vendor RIL library [2]. This flexibility is vital due to the diversity of baseband chipsets used by different manufacturers, necessitating a vendor RIL that is adeptly tailored to the unique characteristics and functionalities of each chipset. These vendor-specific implementations are integrated into a shared library (`.so`). For instance, AOSP provides a reference vendor RIL implementation utilizing the Hayes AT command set [4, 55]. Notably, while vendor RILs are permitted to employ various protocols for baseband communication, they must adhere to the standardized RIL interfaces to ensure compatibility with the generic RIL layer.

In this paper, our primary focus is on these vendor RILs, which present a unique and standard interface, offering a strategic vantage point for reverse engineering the proprietary baseband interfaces. By design, these vendor RIL libraries encompass many undisclosed commands and protocols with the baseband, which are not only critical in understanding the complicated workings of the baseband but also helpful in identifying potential security and privacy vulnerabilities inherent in these systems. Therefore, by dissecting and analyzing the vendor RIL, we can unveil hidden aspects of the baseband, thereby contributing to the broader understanding of its security posture and operational dynamics.

3 Overview

3.1 Objectives and Assumptions

Objectives. Our main objective is to reverse engineer the *vendor implemented RIL libraries* to identify proprietary baseband commands used for communication between AP and CP. To this end, we define two types of such command types based on the direction: (1) *Solicited commands* issued by the AP to instruct the CP for dedicated functions and (2) *Unsolicited commands* used by the CP to proactively interact with the AP via notification mechanisms. These proprietary commands are fundamentally exchanged via standard communication interfaces at the RIL. Essentially, our goal is to comprehensively identify these proprietary command payloads and demonstrate their security impacts.

Assumptions. Considering the availability of RIL firmware and open architecture, this paper focuses on Android RIL [2]. We assume the smartphone vendors have followed the official RIL architecture, which allows us to identify and extract the proprietary vendor RIL libraries from the phone's factory images. We assume our targeted vendor RIL libraries can be disassembled and decompiled, and they are stripped and not obfuscated (which is true at this time of writing). As RIL resides at the native layer of Android, the associated libraries are developed with C++ language for ARM architecture that is widely adopted by most commodity smartphones.

3.2 Running Example

To demonstrate our approach, we use a running example: a solicited command from the AP's vendor RIL querying information from the baseband. This example shown in Figure 2 uses decompiled code from the `libsec-ril.so` library, extracted from a Samsung Galaxy A53 with an Exynos chipset. Here, the `IpcTxCallGetCallList` function constructs the command payload, indicating a request for the call list from the baseband. A virtual function call at line 8 in

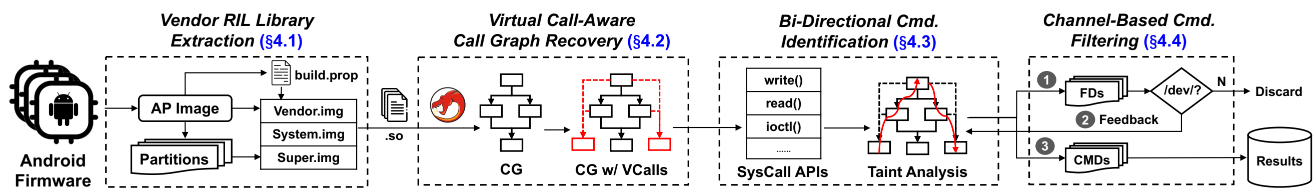


Figure 3: Illustration of the key components of BaseMirror.

`IpcTxCallGetCallList` initiates the command's journey to the baseband. This call uses a function pointer combined with the `this` pointer and offsets, directing control to the `SendMessage` function, where the payload is passed as an argument. For simplification, this example does not explain how the virtual callee is resolved and the explanation is given in §4.2 when we describe how virtual calls are resolved. Following a similar process, the command reaches the `IoChannel::Write` function through another virtual call and is finally sent to the baseband via the Linux system call API `write`. This example focuses on a solicited command, but the process for unsolicited commands, like AP polling requests from the CP, is similar. Unsolicited commands are typically retrieved from `read`-like system calls and processed by appropriate utility functions.

3.3 Challenges and Insights

While the example explains how a specific baseband command is constructed and sent, we need to further generalize these observations to achieve our reverse engineering goals. However, we notice there are three major challenges:

(1) Systematically identifying baseband commands. A mobile device vendor usually defines hundreds of specific baseband commands for various radio functionalities, leading to very complicated implementations. For instance, a vendor RIL library we collected typically occupies a few megabytes of binary code with more than 15K functions. Therefore, manual analysis is extremely inefficient, and thus an automated and systematic approach is highly desired. Such a binary analysis approach should precisely locate the baseband commands and correctly extract their syntax. Simply relying on program heuristics (e.g., function and variable symbols) is not scalable and lacks generality, as it will likely introduce errors when encountering different firmware.

To address this challenge, we observe that the baseband commands within the RIL are issued through standard Linux system calls, such as `write` and `read` as in the running example in Figure 2. Therefore, we design a unified static taint analysis approach [1, 66] to precisely track how baseband commands are constructed from these system calls. Note that we favor a static program analysis approach over dynamic analysis due to scalability concerns, as dynamic analysis requires real devices or emulation capabilities. As a static analysis approach, we only require the RIL binary code, which is readily available from the mobile device firmware.

(2) Efficiently recovering virtual function calls. The first critical step of static taint analysis is to construct a complete Inter-Procedural Call Graph (CG) [41]. An incomplete CG will interrupt the correct program control flow and thus prevent our taint analysis from reaching the desired command locations. In particular, virtual function calls (or *vcalls*) are the major roadblocks as current off-the-shelf program analysis frameworks [42, 48, 54] cannot resolve vcalls by default during call graph construction. These vcalls are ubiquitous in our problem domain due to the flexibility of programming (e.g., various command handlers invoked from a single statement). For example, in Figure 2, the backward command propagation will break at `IoChannel::Write` if we miss the desired virtual function call (`IpcModem::DoIoChannelRoutingTx` to `IoChannel::Write`) from the function call graph.

There have been a handful of program analysis techniques for analyzing virtual function calls at source code level [5] and recovering virtual inheritance at binary level [11, 14, 44]. Our challenge differs in that we are directly solving the concrete vcalls instead of the virtual inheritance. More specifically, when encountering a virtual call site during taint analysis, we need to find the possible callees to propagate the control flow. To this end, we devise a lightweight vcall recovery algorithm that combines class type analysis and offset computation to identify the possible vcall targets and extend the program's control flow.

(3) Accurately filtering undesired commands. After identifying the possible baseband commands using static taint analysis, we still need to filter these results since there could be many false positives. This is due to the generic static analysis sources and sinks (e.g., the `write` system call function) that are not only used for baseband commands but also other undesired purposes. For instance, we notice that some RIL implementations use `read` and `write` over a local file to achieve asynchronous communications. Therefore, we must accurately filter out undesired commands from our results.

Our key insight comes from the general RIL architecture design in §2 that actual interactions between AP and CP are through the Linux-mounted device interfaces, which are distinguished from other types of operations. For example, we notice that both Samsung and Google devices' RIL implementations leverage the mounted devices under the `/dev/` folder. Those devices represent the communication channels to the peripheral devices, including the baseband. Based on this insight, we leverage the command channel represented by the file descriptor (FD) parameters in the `read` and `write` system calls to perform filtering. At a high level, we use backward static analysis to track how the channel FD is initialized and further resolve its path in the system.

4 Design and Implementation

This section presents the design and implementation of BaseMirror, a static analysis tool to identify and extract the baseband commands from the vendor RIL library. Its high-level workflow

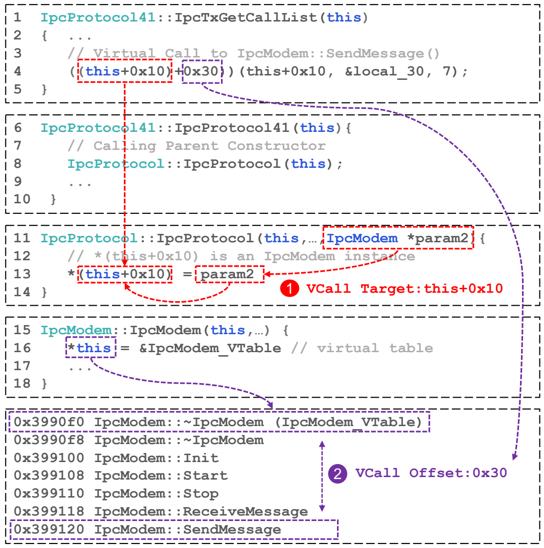


Figure 4: Illustrative example of a virtual function call of our interest in a real vendor RIL library.

Algorithm 1: Virtual Function Call Recovery.

```
Function vcall_recovery(𝔾)
    (V, E) ← 𝔾; I ← ∅; S ← ∅
    foreach instruction i ∈ V do
        if opcode(i) is indirect call then
            I ← I ∪ {i}                                  // Add ins to solved list
    foreach instruction i ∈ I do
        v_c ← class of i
        v_t ← virtual call target expression of operand(i)
        v_o ← virtual call offset of operand(i)
        if (v_c, v_t, v_o) ∈ S then
            continue                                     // vcall already solved
        F ← {v_c's constructors} ∪ {v_c's class member functions}
        foreach function f ∈ F do
            c ← class_inference(v_t, f)
            if c is not null then
                break
        c_vt ← c's virtual function table's base
        c_vf ← c_vt + v_o
        S ← S ∪ {(v_c, v_t, v_o)}                        // Add to solved list
        E ← E ∪ {(f_i, c_vf)}                            // Add vcall to CG
Function class_inference(v_t, f)
    d ← data definition of v_t in f
    if d is a parameter then
        return parameter's class
    else if d is a return variable then
        return return variable type
    else if d is a stack variable then
        return class_inference(d, f)
```

is shown in Figure 3 with four major components. First, taking a vendor-specific Android firmware as input, it automatically unpacks the firmware into multiple partitions and extracts the vendor RIL libraries (§4.1). Second, it disassembles the library and produces a virtual call-aware inter-procedural call graph (§4.2). Third, based on the enhanced CG, it performs bidirectional taint analysis to uncover both solicited and unsolicited commands (§4.3). Lastly, it filters the commands based on their associated communication channels, and those that interact with the baseband are output as results (§4.4).

4.1 Vendor RIL Library Extraction

The initial input of BaseMirror is a batch of Android firmware images implemented by various vendors. BaseMirror uses an automated script to unpack the firmware, examine the partitions, and extract the vendor RIL libraries for later analysis. Specifically, it first extracts the AP image and discards the rest (e.g., CP image), and further divides the AP image into multiple partitions. Due to the diversity of the firmware versions we may handle, it targets three standard Android partitions: `system`, `vendor`, and `super`, which contain system-of-chip (SoC) specific code and libraries. Afterward, each of these partitions will be automatically mounted as file systems for library extraction.

The remaining challenge is to accurately extract the desired vendor RIL libraries of interest. Fortunately, while the naming convention and library path are varied across vendors, we find that there is a systematic way to achieve this goal. Specifically, the `build.prop` file under each partition defines critical system properties and settings, including an important field `vendor.rild.libpath` that points to the exact vendor RIL library location. As such, our script easily pinpoints and extracts the vendor RIL library.

4.2 Virtual Call-Aware Call Graph Recovery

Given a vendor RIL library, BaseMirror disassembles it and generates an inter-procedural call graph (CG). We use the default disassembler from the state-of-the-art static analysis framework Ghidra [42]. However, as mentioned in the challenges, the produced CG is incomplete and we still need to recover indirect function calls especially *Virtual Function Calls*. To provide an in-depth explanation, we provide a zoom-in view of the running example (Figure 2) with more details about the *Virtual Function Calls*.

Figure 4 explains how `IpcTxCallGetCallList` from the example can invoke a virtual call to `IpcModem::SendMessage`. Line 4 reveals the structure of a virtual call which constitutes a virtual call target and a virtual call offset. More specifically, a virtual call target indicates the concrete C++ Class of the instance that initiates the call, and a virtual call offset determines the exact virtual function of that Class. In this example, the virtual call target defined by `this+0x10` is initialized at the parent constructor function (Line 13 of the `IpcProtocol` constructor), as `IpcProtocol41` inherits from `IpcProtocol`. Based on the statement, the virtual call target is initialized by `param2` which suggests a pointer to an `IpcModem` instance based on the parameter type. This further indicates that `this+0x10` (assigned by `param2`) also points to an `IpcModem` instance, and thus the virtual function to be invoked is the `IpcModem` instance's address pluses a constant offset `0x30`. To resolve this function, we notice that the `IpcModem` instance initiates its `this` pointer to the base of its *Virtual Call Table*. Therefore, we can compute the exact virtual function's address based on the constant offset, and it is thus resolved as `IpcModem::SendMessage`. Based on the above example, we present a lightweight virtual call resolution algorithm in Algorithm 1. It is broken down into two major tasks to resolve a virtual call as described below.

(I) Identifying Virtual Call Instructions. Algorithm 1 takes the control flow graph $\mathbb{G} = (V, E)$ of the vendor RIL library as input, and expands the set of edge E with virtual calls. Since our analysis involves taint analysis that may traverse $\mathbb{G}$ in a backward order, E must be complete to ensure the coverage of analysis (while in forward taint analysis, we can solve only encountered virtual calls). Line 1 to Line 5 describes the initial steps to identify virtual call instructions of our interest. Fundamentally, virtual calls in disassembly code are of similar structures as indirect calls [39, 44, 57], as shown in the Figure 4 example. Based on this observation, the algorithm traverses the instructions of each basic block in V. It leverages the GHIDRA's P-Code Intermediate Representation (IR) lifter to convert the disassembly into unified P-Code IR syntax as the binaries of interest are cross-architecture [17]. Based on the P-Code instruction syntax, we consider each instruction i represents a potential virtual function call if it is an indirect call instruction (i.e., with `CALLIND` opcode in P-Code [17]), and if so we add it to the list I.

(II) Resolving Virtual Call Function Target. Line 6 to Line 20 illustrates the steps to resolve the virtual calls for each shortlisted instruction i. First, it obtains the corresponding P-Code representation of the vcall. In our running example Figure 4, the vcall at line 4 of the `IpcTxCallGetCallList` function is represented by `((this+0x10)+0x30))`, which is described by an abstract syntax tree (AST) structure in P-Code [17]. It then parses this representation to extract the vcall table base v_t (e.g., `(this+0x10)` and the vcall table offset v_o (e.g., `0x30`). Since we notice that a Class may reuse its vcalls, we maintain a set S to store and query all vcalls that have already been resolved. To uniquely identify a vcall as a query key, we construct a tuple (v_c, v_t, v_o) using the concrete Class and vcall representation through the following two steps:

- **Virtual Table Analysis**. The first step in our analysis is to identify the concrete Class for a virtual call (vcall) target, determining its virtual table base. This task is challenging in C++ due to difficulties in locating the Class member variable's data definition, which may not be in the same function using the variable. Fortunately, we find that this problem can be solved by limiting the search scope to a list of search functions (F) including all (parent) constructors as well as member functions of the current Class v_c, which may initialize the target through the shard `this` pointer. For example, in Figure 4, the list F includes the parent constructor `IpcProtocol` of the current Class `IpcProtocol41`, which involves the exact definition of `this+0x10`. To implement this searching procedure, we use the abstract syntax tree (AST) representation for P-Code variables [17]. It allows us to find the corresponding data definition by matching the AST representations (e.g., `this+0x10`). Based on the data definition, a `class_inference` procedure (Line 21-Line 28) is introduced to further infer the concrete Class c. Particularly, if the data definition is a function parameter, the corresponding parameter type will be used as the Class type, such as `this+0x10` in Figure 4 is assigned an `IpcModem` type of `param2`. Due to the virtual inheritance in C++, such a Class type can indicate all child Classes (e.g., all Classes that inherit from `IpcModem`). Therefore, it considers the Class hierarchy and returns all possible Classes, which can be inferred from constructors [16].

Analysis Task	API	Parameter Types	Tainted Arg Index
Backward Taint Analysis	`write`	`int`, `void*`, `size_t`	0, 1, 2
	`ioctl`	`int`, `unsigned long`	1
	`sendto`	`int`, `void*`, `size_t`, `int`, ...	1, 2
	`__write_chk`	`int`, `void*`, `size_t`, `size_t`	0, 1, 2
Forward Taint Analysis	`read`	`int`, `void*`, `size_t`	0, 1, 2
	`__read_chk`	`int`, `void*`, `size_t`, `size_t`	0, 1, 2
Command Filtering	`open`	`char*`, `int`	1
	`__open_2`	`char*`, `int`	1
	`fopen`	`char*`, `char*`	0
	`pipe`	`int *`	0

Table 1: Linux system call APIs for BASEMIRROR's analysis.

- **Virtual Function Computation**. Based on the inferred Class of the vcall target, the corresponding virtual function table c_{vt} can be looked up from the static memory region, which includes a list of virtual functions within that Class. By adding the base address of the virtual table c_{vt} with the previously resolved offset v_o, the concrete vcall function c_{vf} is resolved. Ultimately, the vcall represented by (i, c_{vf}) is added to the control flow E.

4.3 Bidirectional Command Identification

After BASEMIRROR recovers a virtual call-enhanced call graph, it performs static taint analysis to identify and extract the concrete command values. Since the commands between AP and CP are either solicited or unsolicited, we employ a bidirectional taint analysis to uncover both commands with different sources and sinks. Based on our observation that the communication between AP and CP must be done via Linux system call APIs, we use these APIs as the sources of taint analysis, as shown in Table 1. In the following, we provide details about our bidirectional analysis.

Backward Taint Analysis. As shown in our previous running example in Figure 2, solicited baseband commands are delivered by system call APIs such as `write` from user space to the kernel space. These `Write`-related APIs are thus the boundary of the data flow that we can trace within the vendor RIL library. Other APIs that represent similar write semantics are also included in Table 1, such as `ioctl` that manipulates device-specific operations. Therefore, the backward taint analysis algorithm starts from these APIs as sources and traces the data flow or the desired API arguments (e.g., the buffer that carries the command payload). Figure 2 highlights the inter-procedural data flow for such an example starting from the `write` API. During the backward control flow transfer across functions, it will query the CG to return both the direct and virtual function calls, to ensure the completeness of our results. To define the ending criterion for the backward taint analysis, we consider the command payload to be either a constant value (for `ioctl` related functions) or a byte array (for `Read` and `Write`-related functions) based on our focused APIs in Table 1. The command shown in Figure 2 is a byte array local variable, which is stored on the function's stack frame. To handle this case, the algorithm emulates the function's execution to recover the stack frame values, thereby obtaining the command value based on its length.

Forward Taint Analysis. In contrast to backward taint analysis, unsolicited commands originating from the CP requires forward

```
FUN_258228(this, long param1)
{   ...
    // Syscall read from baseband
    sVar4 = read(Var9, param1+0x10, 0x40800);
    ...
    IoChannel::DoIoChannelRouting(Var8, param1+0x10, ...);
    ...
}
```

```
IpcModem::DoIoChannelRouting(this, char *param1, ...)
{   ...
    IpcModem::DoIoChannelRoutingRx(this, param1 , ...);
    ...
}
```

```
IpcModem:: DoIoChannelRoutingRx(this, char *param1, ...)
{   ...
    uVar3 = Nv::ProcessRfsPacket(this+0x50, param1, ...);
    ...
}
```

```
Nv::ProcessRfsPacket(this, char *param1, ...)
{   ...
    switch(param1[4]){
     case 1:
       uVar1 = Nv::ProcessNvRead(this);
       return uVar1;
     case 2:
       uVar1 = Nv::ProcessNvWrite(this);
       return uVar1;
    ...
}
```

Figure 5: Illustrative example of a forward taint analysis path for an unsolicited baseband command.

taint analysis. To explain this problem, we provide a motivating example in Figure 5, in which the AP processes the unsolicited baseband command and invokes different handlers according to the command syntax. Similar to the solicited commands, the unsolicited commands are acquired from the system call API `read`. They are then stored in a buffer and passed along various processing functions. Eventually, they reach a dispatcher function `Nv::ProcessRfsPacket` that invokes different handler functions based on the command value. From the descriptions, unsolicited commands are propagated forward after they are read by the vendor RIL. Hence, we define the taint sources to be corresponding `Read`-related system calls, and the sinks to be comparison instructions (e.g., `INT_EQUAL` in P-Code) that represent different branches to dispatch the command. The forward data flow propagation also considers the virtual function calls recovered in the previous step.

4.4 Channel-Based Command Filtering

Before describing the overall command filtering approach, we explain why it is necessary with a motivating example Figure 6. If BaseMirror simply performs taint analysis on all API call sites, the result represents a superset of the solicited and unsolicited baseband commands, i.e., it includes false positive results which are commands not dedicated to the baseband. The counterexample in Figure 6 shows such a case obtained from backward taint analysis from the `write` system call but turns out to be an undesired command. As shown, the buffer of the `write` function backward propagates to the function argument of `StoreStringToFile`, which is invoked to write a string `true` to a *file descriptor (FD)*. Following the `__fd` argument, we can pinpoint its definition from the `open` system call. It is actually an opened file descriptor for a system file at `/efs/imei/selective`. This example inspires us to use the

```
ImeiManager::OnImeiFactoryReset(this,...){
    ...
    ImeiManager::StoreStringToFile("/ets/imei/selective"
                                  ,"true");
    ...
}
```

```
ImeiManager::StoreStringToFile(this, char *param1,
                                     char *param2){
    ...
    __fd = open(param1, 0x42, 0x1fd);
    ...
    // Not a solicited command for baseband
    sVar4 = write(__fd, param2, __n);
    ...
}
```

Figure 6: Motivating example of file descriptor-based filtering, showing a `write` call irrelevant to CP communication.

command channel represented by the file descriptor to filter out false positive results. More specifically, the communication with the baseband is typically conducted through peripheral devices mounted at `/dev/`, such as USB, audio, and the baseband. Therefore, the system path associated with the file descriptor indicates the communication channel of the system calls. Meanwhile, we assume the vendor RIL library solely handles the baseband communication without engaging with other hardware due to its design, and thus we do not further filter the concrete device names under the `/dev/` system path.

Next, we describe the detailed algorithm to perform channel-based filtering in Algorithm 2. Before the taint analysis (§4.3), it first iterates all targeted system API call sites to filter out those not dedicated to the baseband (Line 1-Line 16). To this end, it performs a backward taint analysis to recursively trace the data definition of the file descriptor argument in the targeted system call (e.g., `read` and `write`). Since a file descriptor is returned from certain system calls such as `open` as shown in Figure 6, the taint analysis ends when it reaches a `CALL` instruction (in P-Code IR [17]) that returns the FD. We further classify this function call into three categories. If it is a `pipe` system call function, we discard it. If it is a custom function (e.g., a wrapper function of `open`), it recursively traces the return value in that function. Otherwise, it is an `open` system call and we switch the analysis target to the file system path argument and perform another backward taint analysis. There could be various system call APIs related to `open` as we have listed in Table 1. Finally, the path associated with the FD is solved, and the algorithm uses a regular expression to check whether it is a valid path starting with `/dev/`. It thus filters out those that do not match this criterion and provides feedback to guide the command identification step. Line 17-Line 20 describes the bidirectional taint analysis procedure that occurs after the system API call site targets an FD that potentially communicates with the baseband.

5 Evaluation

We have implemented a prototype of BaseMirror with over 4K lines of Java code based on Ghidra [42]. In support of open science, we have released its implementation on GitHub. In this section, we present our evaluation results. Specifically, we aim to answer the following research questions:

- **RQ1:** How many commands were identified for each firmware?

Algorithm 2: Channel-Based Command Filtering.

```
Function command_filtering(𝔾)
    ℂ ← ∅                                              // Final command set
    foreach target system API call site s do
        // Filter commands with fd analysis result
        i_f ← argument index of fd in s
        f ← taint_analysis(𝔾, s, i_f, 0)               // Backward
        if opcode(f) is not CALL then
            continue
        if operand(f) is open then
            i_f ← file system path arg index
            p ← taint_analysis(𝔾, f, i_s, 0)
            if not match(p, "^/dev/([^/ ]*)+(/[^/ ]*)*?$") then
                continue                               // Discard command
        else if operand(f) is pipe then
            continue                                   // Discard command
        else
            Recursively find fd source
        // Run taint analysis to extract commands
        i_b ← argument indices of cmd buffer in s
        d_b ← taint analysis direction for s
        c ← taint_analysis(𝔾, s, i_b, d_b)
        ℂ ← ℂ ∪ {c}                                    // Add command
    return ℂ
```

- **RQ2:** How to verify the accuracy of the reverse engineered commands and what is their accuracy?
- **RQ3:** What are the semantics of the reverse-engineered commands and how they are distributed?
- **RQ4:** How did the commands evolve over time?

5.1 Experiment Setup

Firmware Collection. Our evaluation with BaseMirror necessitated a comprehensive firmware analysis. To achieve this, we present an extensive examination of 28 Samsung firmware images. Samsung was selected due to its status as one of the most popular and extensively used Android smartphone brands and its firmware is readily available (e.g., from third-party repositories such as SamMobile [49]). However, BaseMirror's analysis methodology is generic and adaptable to other RIL vendors, as discussed in a later section (§7). The collected firmware samples were chosen to cover a wide range of models, regions, and operating system versions. This is to ensure a broad and detailed understanding of the vendor RIL libraries across a spectrum of mobile devices and firmware variations. The firmware metadata is detailed in Table 2. For brevity, we will refer to the vendor RIL libraries using the associated device model names in the table (e.g., **G920** for Galaxy S6).

Experiment Environment. Our evaluation involves 28 firmware images, which were rigorously tested on a Ubuntu 20.04 laptop equipped with an Intel i7-10710U processor and 16GB of RAM. BaseMirror ran on top of Ghidra 9.2.2 and OpenJDK 11.0.2, providing a stable and efficient environment for our analysis. This setup was carefully chosen to optimize the performance and accuracy of our firmware evaluation process. With respect to the running time, it took BaseMirror 4.2 hours to complete the analysis for the firmware we analyzed. Note that we enabled three threads so that it can process three binaries in parallel, and ensure optimal memory utilization efficiency. On average, it took BaseMirror nine minutes to finish analyzing a single binary.

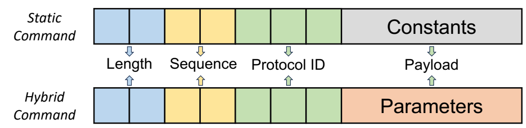


Figure 7: RIL command formats for Samsung Galaxy devices.

5.2 Command Quantity (RQ1)

Classification. Before presenting the statistics, we first discuss the firmware classification methodology. As shown in Table 2, the 28 firmware falls into two categories based on their RIL communication protocols, including 8 firmware using the legacy `Ipc41` protocol and 20 firmware using the latest `Ipc41X` protocol. We were able to identify these firmware attributes based on the program symbols that could not be stripped. For older firmware, all commands are uniformly implemented under the Class `IpcProtocol41` without further detailed differentiation of functional modules. In contrast, newer firmware exhibits a more detailed segmentation of vendor RIL layer functionalities, such as `IpcProtocol41Call` for Phone Call related functions and `IpcProtocol41Sms` for Short Message Service related functions. Consequently, we have unified these diverse implementations under `Ipc41` and `Ipc41X`. This classification approach facilitates later discussion regarding the categorization of different baseband command semantics.

Statistics. Based on the concrete command values, we eventually obtained 873 unique baseband commands from the 28 firmware. These statistics indicate that Samsung heavily reuses these protocol-specific commands among the devices to implement communication between AP and CP. However, we also notice diversified discrepancies among the firmware we have evaluated, which are extensively discussed in the extended version of this work [36]. In Table 2, we break down the command statistics according to the taint sources defined in Table 1. Note that we have aggregated the results of `__write_chk` with `write` as they are both considered the `write` system call, which is also applied to `__read_chk` and `read`. The table indicates that `write` and `read` significantly contribute to the commands compared to `Ioctl` and `Sendto`. Among these two major categories, BaseMirror discovers more `Write`-related baseband commands than `Read`-related ones, indicating the developers tend to integrate more diverse functionalities for AP to CP communication than the opposite direction.

5.3 Command Accuracy (RQ2)

BaseMirror has extracted hundreds of commands between AP and CP. However, verifying the accuracy of these results is necessary and a matter of concern. Despite the challenge of lacking control over the CP both locally and remotely, the correctness of commands sent from CP to AP is relatively easy to confirm due to the partially open-source nature of the AP-side code. Additionally, the community has provided numerous tools for debugging the AP-side code and tracing data. In contrast, the CP-side operates as a complete black box, with limited knowledge and lacking tools for tracking its runtime behavior. Therefore, this section primarily introduces our testing methodologies for the commands from AP to CP.

Firmware Metadata							Evaluation Statistics and Commands per API				
Device Name	Device Model	Firmware	Region	OS Ver.	Ipc41	Ipc41X	#VCall	#Write	#Read	#Ioctl	#Sendto
Samsung Galaxy Note10+	SM-N975F	N975FXXU1ASHE	Egypt	9	●	○	2,828	478	49	41	2
Samsung Galaxy J6	SM-J600FN	J600FNXXU5BSH1	UK	9	●	○	2,476	427	53	40	2
Samsung Galaxy S6	SM-G920F	G920FXXS6ETI6_G920FEVR6ETK1_EVR	UK	7	●	○	3,057	570	67	45	1
Samsung Galaxy S7	SM-G930F	G930FXXU8EUE1_G930FOXA8EUE1_BTU	UK & IRE	8	●	○	2,736	433	53	45	2
Samsung Galaxy S9	SM-G960F	G960FXXUHFVB4_G960FOXMHFVC2_TNZ	New Zealand	10	●	○	3,020	493	55	44	2
Samsung Galaxy S8	SM-G950F	G950FXXU5DSFB	Romania	9	●	○	2,552	445	53	37	2
Samsung Galaxy Note8	SM-N950F	N950FXXUFDUG5_N950FUWEFDUG4_IUS	Mexico	9	●	○	2,551	429	53	39	2
Samsung Galaxy Note9	SM-N9600	N9600ZHU9FVB3_N9600OWC9FVB3_IUS	Mexico	10	●	○	3,517	493	55	62	2
Samsung Galaxy A20	SM-A205F	A205FXXSACVC2_A205FOLMACUI2_MM1	SINGAPORE	11	○	●	3,300	505	204	42	2
Samsung Galaxy A30	SM-A305F	A305FDDU6CUI3_A305FOLM6CUJ1_MM1	SINGAPORE	11	○	●	3,301	505	204	42	2
Samsung Galaxy A40	SM-A405FN	A405FNXXU4CVD1_A405FNOXM4CVD1_EUR	SPAIN	11	○	●	3,302	505	204	42	2
Samsung Galaxy A41	SM-A415F	A415FXXS1CVC1_A415FOXM1CVA3_MEO	Hungary	11	○	●	3,296	505	204	45	2
Samsung Galaxy A50	SM-A505F	A505FDDU9CVC2_A505FOJM9CVC2_LYS	UAE	11	○	●	3,304	505	204	42	2
Samsung Galaxy A60	SM-A606Y	A606YXXU2CVA1_A606YOLO2CVA1_BRI	Taiwan	11	○	●	4,549	505	204	59	2
Samsung Galaxy A70	SM-A705F	A705FXXU5DVC1_A705FOLM5DVB2_XXV	VIETNAM	11	○	●	4,550	505	204	59	2
Samsung Galaxy A80	SM-A805F	A805FXXS6DVD1_A805FOWC6DUJ1_UNE	Mexico	11	○	●	4,549	505	204	59	2
Samsung Galaxy S20 5G	SM-G981B	G981BXXUDFVC7_G981BOXMDFVC7_XSA	Croatia	12	○	●	3,327	525	205	42	2
Samsung Galaxy S21 5G	SM-G991B	G991BXXS4CVCG_G991BOWO4CVB9_IUS	Mexico	12	○	●	3,327	525	205	42	2
Samsung Galaxy Note20 5G	SM-N981B	N981BXXS3FVC8_N981BOXM3FVC5_NZC	UK & IRE	12	○	●	3,327	525	205	42	2
Samsung Galaxy A42 5G	SM-A426B	A426BXXU3DVC2_A426BOXM3DVC2_MM1	ITV	12	○	●	4,580	525	204	59	2
Samsung Galaxy A52 5G	SM-A526B	A526BXXS1CVD1_A526BOLM1CVB6_XSA	Australia	12	○	●	4,580	525	204	59	2
Samsung Galaxy A72	SM-A725F	A725FXXU4BVC1_A725FOJM4BVC1_ILO	ISRAEL	12	○	●	4,580	525	204	59	2
Samsung Galaxy A90 5G	SM-A908B	A908BXXU5EVD2_A908BOXM5EVD2_AUT	Switzerland	12	○	●	4,578	525	204	59	2
Samsung Galaxy A51	SM-A515F	A515FXXU5FVD2_A515FOLM5FVD2_MM1	SINGAPORE	12	○	●	3,323	526	189	36	2
Samsung Galaxy A51 5G	SM-A516B	A516BXXU5DVC2_A516BOXM5DVC2_AUT	SPAIN	12	○	●	3,323	526	189	36	2
Samsung Galaxy S10	SM-G973F	G973FXXUEHVC6_G973FOWCEHVC6_IUS	Mexico	12	○	●	3,320	526	189	36	2
Samsung Galaxy S10 5G	SM-G977B	G977BXXSBHVD1_G977BOXMBHVC6_BTU	UK & IRE	12	○	●	3,325	526	189	36	2
Samsung Galaxy A53 5G	SM-A536E	A536EXXS4AVJ3_A536EOWO4AVI2_ARO	Argentina	12	○	●	3,327	526	189	36	2

Table 2: Metadata of the Android firmware and their evaluation statistics (● indicates the firmware belongs to that category).

To validate the commands, we first classify them into two categories: **Static Commands** and **Hybrid Commands**, as shown in Figure 7. The distinction between the two lies in the fact that hybrid commands include dynamic parameters from the upper layers such as applications, whereas static commands consist of constant values without any parameters. We choose static commands as the dataset for validating the correctness of BaseMirror methodologies because of their parameter-free nature, ensuring the accuracy of the results. However, hybrid commands obtained through static analysis lack runtime parameter information from external sources, and thus cannot be directly validated.

Since these proprietary commands and protocols are not publicly available, it is impossible to compare extracted commands with a ground truth database. To validate the static commands identified, a major challenge is to acquire the command execution results in the modem from the AP side. We observed that after executing the commands, CP sends a response to AP, notifying the execution result. As such, we use a dynamic analysis approach to trigger real commands and intercept and decode response values to determine the correctness of the uncovered commands. More specifically, we developed a testing framework to automate the sending of static commands through the `/dev/umts_ipcX` interface to CP, triggering a variety of behaviors on a Samsung Galaxy A53 device. Meanwhile, to capture the feedback of command execution, we use Frida [47] to hook the response handling APIs in RIL (e.g., `SecRilProxy::OnUnsolicitedResponse`), and check if the response code is in the valid set and whether a non-NULL response has been built. With this approach, we validated 34% of AP to CP commands on the Samsung Galaxy A53, totaling 179 static commands, with no false positives. For the remaining 66% commands, we further discuss the feasible methodology to validate them in §7.

5.4 Command Semantics (RQ3)

Interestingly, our analysis found numerous semantic symbols in the RIL binaries, though they are *not* required by BaseMirror to perform the analysis. These symbols actually come from the *.dynsym* section and thus cannot be stripped due to dynamic linking purposes. To make use of these symbols for command comprehension, we extract the root node function of the taint analysis chain and extract the function name semantics to understand the corresponding command. The intuition is that the root node function that generates the command typically provides the corresponding semantic associated with the command introduced by the developers. As illustrated in Figure 2, the root node function in this taint analysis chain, namely `IpcTxCallGetCallList`, suggests its functionality is to transmit a request to retrieve the list of phone calls. However, other functions along the taint analysis chain, such as `DoIoChannelRoutingTx`, do not exhibit a direct relationship with the command's functionality.

Based on the root node function names, we developed an automatic tool for analyzing the command semantics. First, we employ a well-known tokenization technique in Natural Language Processing (NLP), which breaks the function names into meaningful semantic tokens. Next, we perform a classification step to aggregate the command semantics in order to obtain a high-level view. To this end, building upon the generated tokens, we filter out the prefix tokens and verbs, such as `Ipc` that do not indicate useful command-wise semantics. The first non-filterable word encountered, typically a noun, is then assigned as the functional category. For example, the tokens of the function `IpcTxCallGetCallList` shown in Figure 2, are first filtered to exclude `Ipc`, `Tx`, and `Get`. Ultimately, the function is classified as belonging to the `Call` category. We call each distinctive category a *module* in the below discussion.

Write-related Commands					Read-related Commands				
Protocol	**Module**	**#**	**%**	**Semantics**	**Protocol**	**Module**	**#**	**%**	**Semantics**
Ipc41	Domestic	332	8.81%	Domestic	Ipc41	Nv	152	34.70%	Non-Volatile
	Cfg	256	6.79%	Configuration		Net	17	3.88%	Network
	Net	240	6.37%	Network		Sms	17	3.88%	Short Message Service
	Call	213	5.65%	Call		Qmi	16	3.65%	Qualcomm MSM Interface
	Ss	146	3.87%	Supplementary Services		Ss	15	3.42%	Supplementary Services
	Snd	110	2.92%	Sound		Call	9	2.05%	Call
	Ims	102	2.71%	IP Multimedia Subsystem		Cdma	9	2.05%	Code Division Multiple Access
	Factory	91	2.42%	Factory		Disp	9	2.05%	Display
	Qmi	82	2.18%	Qualcomm MSM Interface		Embms	9	2.05%	Evolved Multimedia Broadcast Multicast Service
	LTE	68	1.80%	Long-Term Evolution		Factory	9	2.05%	Factory
Ipc41X	Domestic	860	8.31%	Domestic	Ipc41X	Cfg	660	16.47%	Configuration
	Net	740	7.15%	Network		Domestic	580	14.47%	Domestic
	Cfg	700	6.77%	Configuration		Nv	380	9.48%	Non-Volatile
	Call	420	4.06%	Call		Call	265	6.61%	Call
	Ss	300	2.90%	Supplementary Services		Sec	160	3.99%	Security
	Snd	260	2.51%	Sound		Net	140	3.49%	Network
	Factory	260	2.51%	Factory		Sap	120	2.99%	SIM Access Profile
	Qmi	220	2.13%	Qualcomm MSM Interface		Sat	120	2.99%	Serving and Tracking Area
	Ims	212	2.05%	IP Multimedia Subsystem		Ss	120	2.99%	Supplementary Services
	Sim	180	1.74%	Subscriber Identity Module		Sim	100	2.50%	Subscriber Identity Module

Table 3: Baseband command semantics categorized by top-10 modules for different protocols.

Semantic Module Distribution. Utilizing the aforementioned methodology, we categorized distinct commands into corresponding functional modules. Then, we conducted a comprehensive analysis of the command frequencies within these modules, with select results presented in Table 3, which delineates the top-10 modules for `Write` and `Read`-related commands in `Ipc41` and `Ipc41X`, and provide the respective command counts and percentages within each module. Upon comparing the numerical counts and proportions, it becomes evident that, despite the version discrepancies between `Ipc41` and `Ipc41X`, the fundamental functionalities remain largely consistent, aligning with our expectations.

Semantics-guided Command Analysis. By semantically clustering commands, we can contrast different command values within the same category, thereby gaining insights into specific bytes of command values. For instance, through comparing command values under the `Sim` and `Domestic` modules, we observed that the fifth byte, `0x05` and `0x20` respectively, serves as the identifier for their modules. By cross-validating with other modules, we found a consistent pattern where the fifth byte across all commands is utilized to denote the command group. This analysis allows us to reverse engineer the proprietary RIL command format in Figure 7. Such information further provides meaningful guidance for dynamic testing such as attack payload discovery discussed later in §6.1. Additionally, the segmentation of command functionalities facilitates a more streamlined analysis, allowing us to focus on modules characterized by a higher command volume and complex functionalities. It optimizes the identification of vulnerable points and potential security issues. In §6, we will expound upon the security issues and potential attacks discovered in Samsung firmware based on our comprehension of command semantics.

5.5 Command Evolution (RQ4)

To answer our RQ4, we provide two high-level overviews of the command evolution with respect to time and device series.

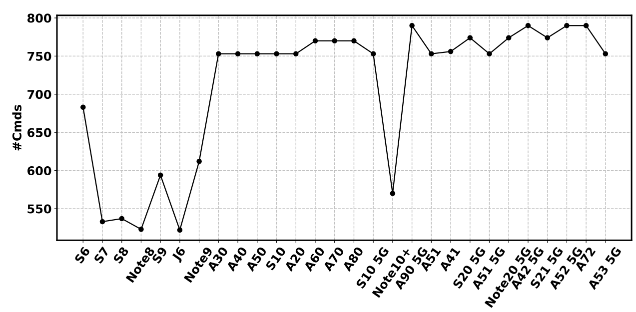


Figure 8: Historical view of baseband command evolution.

Historical View. We first visualize the evolution of baseband command quantities in Figure 8, where the devices are ordered chronologically per their release date (ranging from 2015 to 2022). The figure shows a growing trend in the number of commands over time, indicating that Samsung has continuously integrated new features into the baseband. Notably, we notice a substantial increase (over 20%) of command numbers after Note 9 was released since the devices afterward have evolved to the `Ipc41X` protocol that brings in significant new commands. More specifically, we found that the new `Ipc41X` protocol splits the original functions in `Ipc41` into more subdivided functional areas, such as `IpcProtocol41Call` and `IpcProtocol41Sms`. This also implies the architecture of CP firmware is likely to have been refactored. Additionally, 5G device models tend to include more baseband functions than non-5G devices, and there is also one interesting exception (Note 10+) in which BaseMirror only reported less than 600 commands, far less than other devices released in its time frame due to the legacy `Ipc41` implementation in the 4G version.

Device Series View. We further provide a different perspective regarding the device series in Figure 9. The figure categorizes the results into three device series, namely series A (15 models), series Note (4 models), and series S (8 models). The J series model is

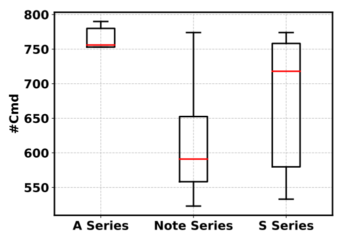
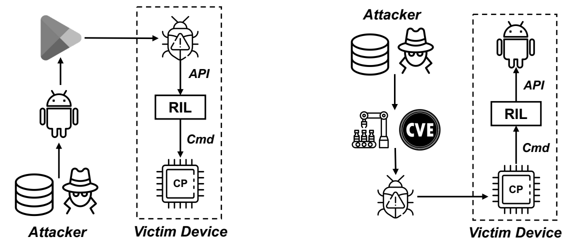


Figure 9: Device series view of command statistics.



(a) Attacks targeting CP. **(b) Attacks targeting AP.**

Figure 10: Workflow of two attacks exploiting the baseband commands from our reverse engineered results.

excluded since there is only one such firmware in our dataset. Our visualization suggests obvious discrepancies in the number of commands among these three series. Specifically, series A models tend to involve more commands since all of them have evolved to the latest `Ipc41X` protocol. In contrast, the Note series and S series involve many legacy models that use the old `Ipc41` protocol.

6 Attacks via RIL

To evaluate the security implication of the extracted commands, this section presents how we are able to construct attack payloads and demonstrate attacks based on our results and findings in §5. We present an automatic attack payload discovery framework, followed by examples of attacks initiated from either the AP or CP on a Samsung A53 5G device (model SM-A536E in Table 2).

Threat Model. We consider that the identified commands from the vendor RIL libraries could be exploited in various ways to compromise user security and privacy. First, regarding *AP to CP* attackers (shown in Figure 10a), they could inject malicious solicited baseband commands through RIL to cause substantial impact such as service disruption without the user's consent. To this end, we consider three distinctive adversary classes with different privileges:

(1) **Privileged Attackers**. Since the command injection through RIL is protected by Android's permission mechanism by default [2], we first consider privileged attackers who have root access or are within the radio group. Such an attacker model may be achieved by examples such as a Remote Access Trojan (RAT) [65] or malicious system applications (e.g., a dialer app).

(2) **Unprivileged Attackers Exploiting Unprotected APIs**. Interestingly, we found that the interaction with RIL can be achieved by unprivileged attackers. This adversary class involves any unprivileged user-space applications, which can be readily introduced to the victim device through app stores and social engineering. These applications further escalate their privileges by exploiting unprotected Android framework APIs, such as secret menus [21], backdoors, and app-accessible system function calls (e.g., `invokeOemRilRequestRaw`) [52], which allows the injection of arbitrary RIL payloads to baseband.

(3) **Unprivileged Attackers Exploiting Unprotected Communication Interfaces**. An alternative way for a user-space malicious program to escalate its privilege is through unprotected communication interfaces. One notable example is the unprotected UNIX sockets implemented in Samsung Galaxy devices that allow any applications to interact with the RILD and inject arbitrary RIL messages to the CP [43].

Regarding *CP to AP* exploitation (depicted in Figure 10b), it requires an adversary with high privilege, e.g., a compromised baseband, kernel, or the OS, that injects malicious RIL commands to the AP through RIL. This could be achieved through existing RCE exploits [18, 23, 40, 67] or supply chain adversaries.

6.1 Attack Payload Discovery

As discussed in §5.3, observing the behavior of commands sent from AP to CP is much more challenging than the opposite direction. Consequently, based on the identified command in §5, we develop an automatic black-box testing framework to discover and validate command payloads exploitable for AP to CP attacks, based on existing debugging tools such as the Android Debug Bridge (ADB). We then applied this framework to a Samsung Galaxy A53 device. In particular, we focus on crash-related attacks, as they can be easily observed with practical security implications.

Our framework extends the testing tool in §5.3 with two additional modules: **States Checking Module** and **Parameters Mutation Module**. Specifically, in the States Checking Module, we compare the mobile device's state before and after command execution, checking if ADB is connectable and if the baseband modem is still in service, to assess the potential malicious exploitation of certain commands. Additionally, inspired by popular fuzzing works [8, 15, 34, 35, 53], the Parameters Mutation Module mutates bytes representing run-time parameters within hybrid commands and uses various response values and status as guidance, to identify values triggering anomalies on the device.

States Checking Module. The States Checking Module quires the `telephony.registry` interface's return value, specifically `mServiceState`. This process determines whether the modem is affected by the command, and assesses the extent of impact. After injecting a command to CP, it checks the state of the modem immediately. If the modem recovers shortly after a transient `OUT_OF_SERVICE` state, it categorizes the command as a *Temporary Crash Command*, indicating that it could at least disrupt the modem's service shortly. Subsequently, the device is rebooted, and the modem state is queried again. If the modem has returned to an `IN_SERVICE` state, the module labels this command as a *Recoverable Crash Command*, indicating that manual intervention can restore service. However, if the modem remains in an `OUT_OF_SERVICE` state, indicating that the command has permanently damaged the firmware's functionality, it is classified as a *Permanent Crash Command*.

Parameters Mutation Module. For hybrid commands, we port AFL++ 's seed mutation mechanism [15] to mutate their corresponding parameter bytes. Additionally, we utilize response values and modem states as guidance to indicate interesting mutation results.

6.2 Attacks Targeting CP

Attack Goal and Procedure. As shown in Figure 10a, the attacker aims at compromising the CP through a malicious AP application via RIL, such as disrupting the cellular service and performing denial-of-service attacks. To begin with, the attacker needs to acquire sufficient knowledge about the attack command payload for the victim device. The desired attack payload could either be static RIL commands that trigger expected outcomes (e.g., power off the baseband), or hybrid commands with malicious parameters (e.g., to set invalid baseband channels). For instance, by running the automated attack payload discovery tool, we show that an attacker can identify 7 exploitable RIL commands (in Table 4) for a Samsung Galaxy A53 model, which takes two weeks to complete. To launch the attack, the attacker implants malicious logic as a privileged or user-space application to be installed on the victim device, which injects the malicious commands to the baseband at run-time.

Security Implication. Table 4 summarizes the 7 commands that can be leveraged to compromise a Samsung Galaxy A53 baseband, which leads to different degrees of impact based on our repeated experimentation. Among them, 3 commands can cause *Temporary Crashes* to the baseband, which recovers after a few seconds. In order to persist the attacks, one can constantly replay these attack payloads to cause service disruption. In addition, we found one command that causes *Recoverable Crash*, and the victim will have to reboot the device to obtain cellular service. In extreme cases, we found three commands that trigger *Permanent Crashes*, which requires a complete firmware re-flash to recover the baseband. We provide the exploit details in the extended version of this work [36].

Root Cause Analysis. We speculate the root causes for the aforementioned attacks are two-fold. First, static RIL commands are essentially abused for malicious purposes as they represent inherent baseband functionalities. For instance, in Table 4, we list the corresponding command semantics extracted from the symbols, including two static commands that reset and power off the baseband. Moreover, for hybrid RIL commands, we speculate that the CP-side parameter checks for command parameters are insufficient, leading to crashes (e.g., segmentation faults) on the CP. Based on the command structures and semantics, these hybrid commands tend to encapsulate complex parameters (e.g., APDU and SMS payloads) that require sophisticated decoding logic to handle them properly.

6.3 Attacks Targeting AP

Attack Goal and Procedure. Based on the analysis of `Read`-related commands, we have identified that the `Nv` Module contains a set of functions accessing the file system on the AP, such as `ProcessNvWrite` and `ProcessNVRead`. In the presence of a compromised baseband modem, these baseband functionalities can be exploited for unauthorized file system access from the high-privileged CP. To this end, the baseband firmware code may initiate a malicious unsolicited command request to the AP via RIL, by embedding the target file path as the command parameter. Notably, while this vulnerability was first reported in 2014 as a backdoor [28], it still exists in a widespread of the device firmware we have analyzed. Meanwhile, we discovered an undisclosed vulnerability in `Nv::ProcessOpenFile` due to the absence of symbolic link checking, which further allows arbitrary file access on the AP.

Security Implication. Typically, due to the invocation of vendor RIL libraries by the `RILD` process, which possesses high system privileges, read and write operations are granted to the attacker on the majority of files. In an existing exploit demonstration by the Replicant project [28], it is shown that an attacker can inject malicious logic to the modem kernel driver (equivalent to an attacker CP), which is capable of opening and reading a user file located under the AP's `/data/` folder. To exploit this vulnerability further, our new finding shows that a malicious symbolic link could be provided as input to enable arbitrary file access on the AP. We provide additional details about this vulnerability in the extended version of this work [36].

Root Cause Analysis. The unsolicited commands within the `Nv` module are possibly designed for the CP to read and write certain logs to the AP's file system, such as dumping the baseband logs. However, since the command allows to specify the file paths within the AP, it enables arbitrary file system access from a malicious CP. Moreover, we also explored the possibility of using `../` for directory traversal attacks (to escape the prefix paths) and found that the firmware has eliminated them with corresponding sanitizer logic.

7 Discussion

Coverage of Validated Commands. As discussed in §5.3, a substantial portion (66%) of the baseband commands discovered involve parameters from external sources, which cannot be directly validated via our framework. Examples of such include heterogeneous inputs from user-space applications, file systems, and networks, such as SMS, configuration files, as well as APDU payloads. These hybrid commands would be useful for further security analysis. For instance, the structural information of these parameter values is useful for dynamic analysis, such as the attack payload discovery process discussed in §6.1, thereby enabling us to identify more exploitable commands. Further, a grey-box fuzzing tool could be developed with parameter-aware mutation mechanisms. However, it is challenging to construct these command payloads with valid parameters. In the following, we categorize them into three types and discuss feasible ways to validate them in future work and show the corresponding examples in the extended version of this work [36], for readers of interest.

(1) **Direct Input Parameter**. We found many parameters directly come from external input sources to configure the baseband, and one such notable example is presented in Figure 13 in the extended version [36] of this work. These parameters are typically of primitive types (e.g., boolean and integer) and are easy to handle, as they directly contribute to the command payload

Crash Type	Command Semantics	Cmd Type	Command Payload (Bytes)
Temporary	IpcProtocol41Power::IpcTxResetOemModem	Static	07, 00, 00, 00, 01, 03, 05
	IpcProtocol41Sap::IpcTxGetSapTransferApdu	Hybrid	0e, 01, 00, 00, 12, 04, 02, ...
	IpcProtocol41Imei::IpcTxImeiPreconfigSet	Hybrid	17, 00, 00, 00, 10, 03, 03, ...
Recoverable	IpcProtocol41Power::IpcTxModemPowerOff	Static	07, 00, 00, 00, 01, 02, 01
Permanent	IpcProtocol41Net::IpcTxSetSystemSelectionChannels	Hybrid	20, 00, 00, 00, 08, 07, 03, 01, ff (x8) ...
	IpcProtocol41Domestic::IpcTxDomesticSetChannelSettingLte	Hybrid	09, 00, 00, 00, 20, 64, 03, ...
	IpcProtocol41Domestic::IpcTxDomesticGetNsriDecryptSms	Hybrid	99, 00, 00, 00, 20, 91, 02, ...

Table 4: RIL command payloads uncovered by the attack discovery framework and the crash types they could trigger.

through simple operations (e.g., concatenation). We have extended BaseMirror and our fuzzer to support this type of hybrid command by marking payload bytes as static or dynamic during taint analysis and validated an additional 16% of AP to CP commands on the Samsung Galaxy A53.

(2) **Derived Parameter**. Unlike the first category, a derived parameter needs to be transformed (e.g., through splitting, reassembly, or conditional evaluation) before it is used as part of the command payload. As these parameters involve computation and are condition-dependent, we consider using symbolic execution [54] to track each byte of the parameter and determine its constraints. Once the constraints are identified, fuzzing can be employed to explore the correct commands. We present one such example in Figure 14 of the extended version of this work [36],

(3) **Structured Parameter**. We also found that many parameters are constructed as certain data structures. An example of such can be a JSON payload involving network scanning information to be transmitted to the baseband. Due to the complexity of these data structures and lack of constraint conditions in JSON files, direct exploration using fuzzing would be highly challenging. Therefore, a more feasible way is to obtain input seeds for the fuzzer, such as dynamically intercepting valid command payloads from normal traffic via Frida [47]. However, such an approach needs to properly trigger these interactions, which still remains a challenge. We further present and discuss a typical example in Figure 15 of the extended version of this work [36].

Generality of Methodology. While in this paper we focus on Samsung's RIL libraries, BaseMirror's methodology is generic and could be applied to other Android RIL vendors such as Qualcomm and MediaTek. This is due to the standard RIL architecture in Android as presented in Figure 1 [2], where the vendor RIL libraries must rely on universal Linux system calls (e.g., `read` and `write`) to interact with the baseband regardless of the vendors. We consider extending BaseMirror for other device vendors as an important future work to unveil additional security issues across a diverse range of devices on the market.

Responsible Disclosure. We have reported our findings to Samsung Security, providing the exploit prerequisites and Proof of Concept (PoC). As of this writing, Samsung has confirmed our findings and patched the vulnerabilities, and awarded us a bug bounty for the lack of symbolic link checking bug within their RIL implementation.

8 Related Work

Baseband Security. The baseband, managing vital cellular functions yet being closed-source and complex, holds significant security interest for both attackers and researchers. There have been several remote code execution (RCE) exploits due to baseband memory corruption [19, 20, 59], accompanied by guides on analyzing and debugging baseband code [6, 18]. Recently, automated static analysis has been used to detect deviations in baseband code from cellular standards [22, 30, 31], uncovering memory bugs. Furthermore, advancements in dynamic firmware analysis and rehosting have facilitated baseband fuzzing [23, 26, 40]. However, research on the RIL remains limited. Previous studies mainly leveraged RIL for cellular traffic monitoring [37, 56] and attack mitigation [63]. ARIstoteles represents a closely related work that scrutinizes Apple's baseband interfaces and uses fuzzing to identify security flaws [32]. However, our research targets Android RIL, a completely different ecosystem that allows baseband vendor-specific implementations.

Mobile Security. Orthogonal to our work, other prior research has delved into various critical components of the mobile system firmware. A few studies target the pre-installed applications in Android OS firmware, uncovering privilege escalation [12] and over-the-air update vulnerabilities [7]. Some extract proprietary AT commands and use fuzzing to understand their security implications [29, 55]. Note that the commands we reverse engineered are not AT commands as they do not have the prefix `AT` [29]. In addition, vendor-specific private APIs [10] and access policies [24] have been identified and raised concerns regarding access control vulnerabilities. To comprehensively measure the Android firmware security, some large-scale longitudinal studies were conducted [25, 46].

C++ Binary Analysis. Binary analysis of C++ has been challenging due to various sophisticated mechanisms. Previous work mainly tackles class hierarchy reconstruction [13, 44], class and object recovery [27, 51], and employing static analysis to uncover bugs and vulnerabilities [50, 61]. In particular, virtual inheritance is a unique challenge in object-oriented languages such as C++. There are techniques to recover virtual inheritance [14] and defenses to protect the control flow integrity of virtual function calls [11, 45]. BaseMirror addresses the challenge of recovering virtual function calls and employs static taint analysis for command extraction.

9 Conclusion

This paper presents a novel approach to unveil security vulnerabilities of basebands from the Radio Interface Layer in the Android ecosystem. Based on this insight, we develop BaseMirror, an automatic tool employing static binary code analysis to reverse engineer vendor-proprietary baseband commands from the vendor RIL binaries. It addresses multiple technical challenges of binary analysis including the recovery of C++ virtual function calls as well as the comprehensive identification and accurate filtering of baseband commands. We have applied BaseMirror to 28 vendor RIL libraries for Samsung mobile devices, and identified 873 unique baseband commands. To evaluate the security implications of our results, we further demonstrated 8 exploitable attacks with the discovered commands on a Samsung Galaxy A53 device, which can interfere with the baseband's cellular service and allow arbitrary access to the AP's file system.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive feedback. This research was supported in part by ARO award W911NF2110081, DARPA award N6600120C4020, and NSF awards CNS-2112471 and ITE-2326882. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the ARO, DARPA and NSF.

References

[1] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*.
[2] Android. 2022. RIL Refactoring | Android Open Source Project. https://source.android.com/devices/tech/connect/ril.
[3] Android. 2024. Android Interface Definition Language (AIDL). https://developer.android.com/guide/components/aidl.
[4] Android. 2024. Android Open Source Project. https://source.android.com/.
[5] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*.
[6] David Berard and Vincent Fargues. 2020. How to design a baseband debugger. In *Information and Communication Technology Security Symposium (SSTIC)*.
[7] Eduardo Blázquez, Sergio Pastrana, Álvaro Feal, Julien Gamba, Platon Kotzias, Narseo Vallina-Rodriguez, and Juan Tapiador. 2021. Trouble over-the-air: An analysis of fota apps in the android ecosystem. In *42nd IEEE Symposium on Security and Privacy (SP)*.
[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[9] Denis 'GNUtoo' Carikli. 2021. Samsung-ipc. https://redmine.replicant.us/projects/replicant/wiki/Samsung-ipc.
[10] Zeinab El-Rewini and Yousra Aafer. 2021. Dissecting residual APIs in custom android ROMs. In *28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[11] Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. 2017. Strict virtual call integrity checking for C++ binaries. In *17th ACM on Asia Conference on Computer and Communications Security (ASIACCS)*.
[12] Mohamed Elsabagh, Ryan Johnson, Angelos Stavrou, Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2020. FIRMSCOPE: Automatic uncovering of Privilege-Escalation vulnerabilities in Pre-Installed apps in android firmware. In *29th USENIX Security Symposium (USENIX Security)*.
[13] Rukayat Ayomide Erinfolami and Aravind Prakash. 2019. DeClassifier: Class-Inheritance Inference Engine for Optimized C++ Binaries. In *19th ACM Asia Conference on Computer and Communications Security (ASIACCS)*.
[14] Rukayat Ayomide Erinfolami and Aravind Prakash. 2020. Devil is virtual: Reversing virtual inheritance in C++ binaries. In *27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Conference on Offensive Technologies (WOOT)*.
[16] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: approaching C++ decompilation. In *18th Working Conference on Reverse Engineering (WCRE)*.
[17] Ghidra. 2024. P-Code. https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/package-summary.html.
[18] Nico Golde and Daniel Komaromy. 2016. Breaking Band: reverse engineering and exploiting the shannon baseband. In *2016 Recon*.
[19] Marco Grassi and Xingyu Chen. 2021. Over The Air Baseband Exploit: Gaining Remote Code Execution on 5G Smartphones. In *BlackHat USA*.
[20] Marco Grassi, Muqing Liu, and Tianyi Xie. 2018. Exploitation Of A Modern Smartphone Baseband. In *BlackHat USA*.
[21] Skanda Hazarika. 2023. Android hidden codes: All the custom dialer codes and what they do. https://www.xda-developers.com/android-secret-codes/.
[22] Grant Hernandez and Kevin RB Butler. 2019. Basebads: Automated security analysis of baseband firmware. In *12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.
[23] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. 2022. FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *29th Network and Distributed System Security Symposium (NDSS)*.
[24] Grant Hernandez, Dave Jing Tian, Anurag Swarnim Yadav, Byron J Williams, and Kevin RB Butler. 2020. BigMAC:Fine-Grained Policy Analysis of Android Firmware. In *29th USENIX Security Symposium (USENIX Security)*.
[25] Qinsheng Hou, Wenrui Diao, Yanhao Wang, Xiaofeng Liu, Song Liu, Lingyun Ying, Shanqing Guo, Yuanzhi Li, Meining Nie, and Haixin Duan. 2022. Large-scale security measurements on the android firmware ecosystem. In *44th International Conference on Software Engineering (ICSE)*.
[26] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. 2021. Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices. In *28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[27] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya Narasimhan. 2014. Recovering C++ objects from binaries using inter-procedural data-flow analysis. In *4th ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW)*.
[28] Paul K. 2014. Replicant developers find and close Samsung Galaxy backdoor. https://www.fsf.org/blogs/community/replicant-developers-find-and-close-samsung-galaxy-backdoor.
[29] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2019. Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones. In *35th Annual Computer Security Applications Conference (ACSAC)*.
[30] Eunsoo Kim, Min Woo Baek, CheolJun Park, Dongkwan Kim, Yongdae Kim, and Insu Yun. 2023. BASECOMP: A Comparative Analysis for Integrity Protection in Cellular Baseband Software. In *32nd USENIX Security Symposium (USENIX Security)*.
[31] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. 2021. BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols. In *28th Network and Distributed System Security Symposium (NDSS)*.
[32] Tobias Kröll, Stephan Kleber, Frank Kargl, Matthias Hollick, and Jiska Classen. 2021. ARIstoteles–Dissecting Apple's Baseband Interface. In *26th European Symposium on Research in Computer Security (ESORICS)*.
[33] laforge. 2019. Qualcomm Linux Modems by Quectel & Co. https://osmocom.org/projects/quectel-modems/wiki/QMI.
[34] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. 2021. From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware. In *28th Network and Distributed System Security Symposium (NDSS)*.
[35] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *44th International Conference on Software Engineering (ICSE)*.
[36] Wenqiang Li, Haohuang Wen, and Zhiqiang Lin. 2024. BaseMirror: Automatic Reverse Engineering of Baseband Commands from Android's Radio Interface Layer. In *axXiv preprint arXiv:2409.00475*.
[37] Yuanjie Li, Chunyi Peng, Zengwen Yuan, Jiayao Li, Haotian Deng, and Tao Wang. 2016. Mobileinsight: Extracting and analyzing cellular network information on smartphones. In *22nd Annual International Conference on Mobile Computing and Networking (MobiCom)*.
[38] Yiming Liu, Cen Zhang, Feng Li, Yeting Li, Jianhua Zhou, Jian Wang, Lanlan Zhan, Yang Liu, and Wei Huo. 2024. Semantic-Enhanced Static Vulnerability Detection in Baseband Firmware. In *46th International Conference on Software Engineering (ICSE)*.
[39] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[40] Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. Basesafe: Baseband sanitized fuzzing through emulation. In *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.
[41] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary hunting with inter-procedural control flow. In *9th International Conference on Information Security and Cryptology (ICISC)*.
[42] NSA. 2024. Ghidra. https://ghidra-sre.org/.
[43] Roberto Paleari. 2024. Interacting with Samsung radio layer (RILD). http://roberto.greyhats.it/2016/05/samsung-access-rild.html.

[44] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs.. In *24th Network and Distributed System Security Symposium (NDSS)*.

[45] Andre Pawlowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2019. VPS: excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching. In *35th Annual Computer Security Applications Conference (ACSAC)*.

[46] Andrea Possemato, Simone Aonzo, Davide Balzarotti, and Yanick Fratantonio. 2021. Trust, but verify: A longitudinal analysis of Android OEM compliance and customization. In *42nd IEEE Symposium on Security and Privacy (SP)*.

[47] Ole André Vadla Ravnås. 2024. Frida - A world-class dynamic instrumentation toolkit. https://frida.re/.

[48] Hex Rays. 2024. IDA Pro. https://www.hex-rays.com/idapro.

[49] SAMMOBILE. 2024. SamMobile - Your source for all Samsung news. https://www.sammobile.com/.

[50] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for C/C++. In *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[51] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. 2018. Using logic programming to recover C++ classes and methods from compiled executables. In *25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[52] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *23th Network and Distributed System Security Symposium (NDSS)*.

[53] Jiameng Shi, Wenqiang Li, Wenwen Wang, and Le Guan. 2024. Facilitating Non-Intrusive In-Vivo Firmware Testing with Stateless Instrumentation. In *31st Network and Distributed System Security Symposium (NDSS)*.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. SOK:(State of) The Art of War: Offensive techniques in binary analysis. In *37th IEEE Symposium on Security and Privacy (SP)*.

[55] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. 2018. Attention spanned: Comprehensive vulnerability analysis of AT commands within the android ecosystem. In *27th USENIX Security Symposium (USENIX Security)*.

[56] Narseo Vallina-Rodriguez, Andrius Auçinas, Mario Almeida, Yan Grunenberger, Konstantina Papagiannaki, and Jon Crowcroft. 2013. RILAnalyzer: a comprehensive 3G monitor on your phone. In *13th Internet Measurement Conference (IMC)*.

[57] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *37th IEEE Symposium on Security and Privacy (SP)*.

[58] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. 2019. Looking from the mirror: Evaluating IoT device security through mobile companion apps. In *28th USENIX Security Symposium (USENIX Security)*.

[59] Ralf-Philipp Weinmann. 2012. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. In *6th USENIX conference on Offensive Technologies (WOOT)*.

[60] Haohuang Wen, Qi Alfred Chen, and Zhiqiang Lin. 2020. Plug-N-Pwned: Comprehensive vulnerability analysis of OBD-II dongles as a new Over-the-Air attack surface in automotive IoT. In *29th USENIX Security Symposium (USENIX Security)*.

[61] Haohuang Wen and Zhiqiang Lin. 2023. Egg hunt in Tesla infotainment: a first look at reverse engineering of Qt binaries. In *32nd USENIX Security Symposium (USENIX Security)*.

[62] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *27th ACM SIGSAC conference on computer and communications security (CCS)*.

[63] Haohuang Wen, Phillip Porras, Vinod Yegneswaran, and Zhiqiang Lin. 2023. Thwarting Smartphone SMS Attacks at the Radio Interface Layer. In *30th Network and Distributed System Security Symposium (NDSS)*.

[64] Haohuang Wen, Qingchuan Zhao, Qi Alfred Chen, and Zhiqiang Lin. 2020. Automated cross-platform reverse engineering of CAN bus commands from mobile apps. In *27th Network and Distributed System Security Symposium (NDSS)*.

[65] wishihab. 2018. Remote Access Tool Trojan List - Android. https://github.com/wishihab/Android-RATList.

[66] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on android with static taint analysis. In *3rd World Congress on Software Engineering (WSSE)*.

[67] Yuanyuan Zeng, Kang G Shin, and Xin Hu. 2012. Design of SMS commanded-and-controlled and P2P-structured mobile botnets. In *5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*.