FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware

Grant Hernandez*¶, Marius Muench†¶, Dominik Maier‡, Alyssa Milburn†,
Shinjo Park‡, Tobias Scharnowski§, Tyler Tucker*, Patrick Traynor*, Kevin R. B. Butler*

*University of Florida, {grant.hernandez, tylertucker1, traynor, butler}@ufl.edu
†Vrije Universiteit Amsterdam, {m.muench, a.a.milburn}@vu.nl
‡TU Berlin, {dmaier, pshinjo}@sect.tu-berlin.de
§Ruhr-Universität Bochum, tobias.scharnowski@rub.de

***Abstract*—Smartphones today leverage baseband processors to implement the multitude of cellular protocols. Basebands execute firmware, which is responsible for decoding hundreds of message types developed from three decades of cellular standards. Despite its large over-the-air attack surface, baseband firmware has received little security analysis. Previous work mostly analyzed only a handful of firmware images from a few device models, but often relied heavily on time-consuming manual static analysis or single-function fuzzing.**

To fill this gap, we present FirmWire, the first full-system emulation platform for baseband processors that executes unmodified baseband binary firmware. FirmWire provides baseband-specific APIs to easily add support for new vendors, firmware images, and security analyses. To demonstrate FirmWire's scalability, we support 213 firmware images across 2 vendors and 9 phone models, allowing them to be executed and tested. With these images, FirmWire automatically discovers and bridges internal baseband APIs, allowing protocol messages to be injected with ease. Using these entry points, we selected the LTE and GSM protocols for fuzzing and discovered 7 pre-authentication memory corruptions that could lead to remote code execution – 4 of which were previously unknown. We reproduced these crashes over-the-air on real devices, proving FirmWire's emulation accuracy. FirmWire is a scalable platform for baseband security testing and we release it as open-source to the community for future research.

I. Introduction

Cellular protocols, and the billions of devices that use them, have ushered widespread connectivity and mobility to the world. Modern-day smartphones use a dedicated *Baseband Processor (BP)*, which is responsible for processing the complex cellular protocol message formats, state machines, timers, and more. These basebands run high-performance firmware, usually driven by a Real-Time Operating System (RTOS).

Unfortunately, analyzing and testing the security of BPs is extremely difficult for outside researchers, as their firmware is typically not only proprietary but also tremendously complex and executed in unfavorable environments (e.g. secure mobile devices) for debugging and introspection. Initial industry-driven security research [6], [21], [54], [67] has demonstrated that over-the-air exploitation of basebands is not only possible but also practical – even remotely – due to the lack of hardening compared to mobile operating systems like Android or iOS.

Recent academic work attempts to automate the processes of baseband security analysis with varying degrees of success. A first approach [40] treated User Equipment (UE) and basebands themselves as black boxes while carrying out automated over-the-air testing. While automated, this approach does not scale as physical radio equipment and many mobile phones are required. Lack of visibility into the baseband's internal state means that crashes, which can indicate a security vulnerability, are at best diagnosed as a "denial of service".

In an attempt to address this problem, a new line of research about automated analysis of *binary* firmware images for BPs emerged. These new approaches either emulate and fuzz test individual parsers embedded in baseband firmware (e.g., [45], [22]) or statically analyze specific components of the firmware, such as control plane layer-3 protocols [38]. The first approach fails to capture interactions with other portions of the firmware and thus does not scale to the hundreds of thousands of functions present in modern baseband firmware. The second suffers from many false positives. Although both approaches found vulnerabilities, they cannot identify more complex issues arising from the interaction of the different components and stacks embedded in modern BPs. This is because neither considers the execution environment of a baseband, ignoring most of the firmware, including the underlying RTOS.

In this paper, we present FirmWire, a baseband emulation platform that addresses the complex and highly stateful nature of baseband firmware. FirmWire demonstrates that full-system emulation of baseband firmware is not only possible but also scalable. It overcomes the limitations of current automated baseband security testing.

In summary, our contributions are as follows:

- **We design and implement FirmWire, a baseband analysis platform,** enabling extensible, scalable, and automated dynamic analysis for 213 baseband firmware images across 2 vendors spanning 2

¶These two authors contributed equally.

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23136
www.ndss-symposium.org

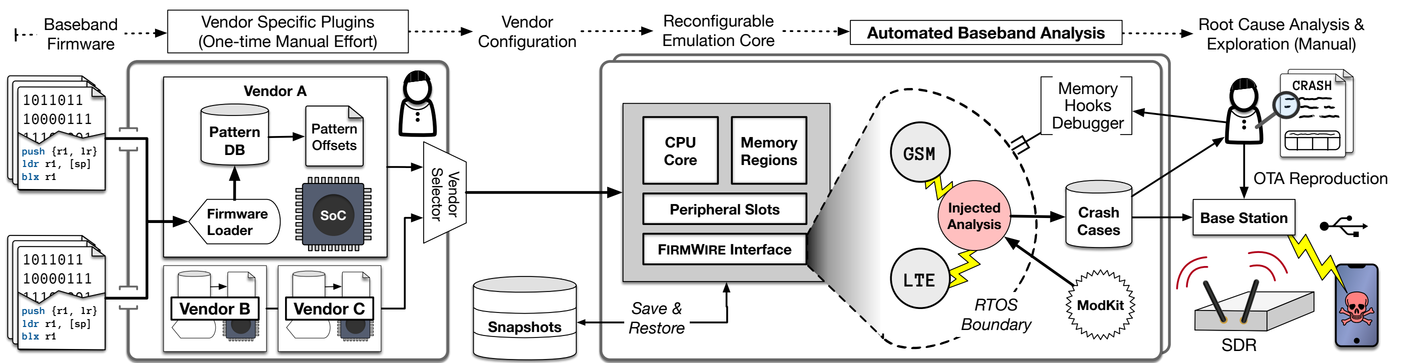


Fig. 1: An overview of the FirmWire baseband analysis platform. Starting from left to right: FirmWire loads the baseband firmware. The vendor plugin abstracts away image-specific details using the vendor firmware loader and PatternDB. Next, the extracted information is used to configure the emulation core. During emulation FirmWire uses a built-in interface to interact with the running firmware, enabling automated dynamic analysis and manual inspection. Finally, discovered crashes can be interactively root caused and verified over-the-air using a base station.

different Instruction Set Architectures (ISAs) and 9 device models. To support additional baseband research and enable extension to other platforms, we release FirmWire's source code and artifacts at: https://www.github.com/FirmWire/FirmWire.

- **Using FirmWire we are the first to emulate Samsung's and MediaTek's closed source proprietary basebands from boot** while providing accurate RTOS execution, fuzzing, and interactive and automatic dynamic analyses. Moreover, we are the first to assess modern MediaTek basebands running on MIPS processors.
- **We discover 7 vulnerabilities, 4 of which are previously unknown and reachable pre-authentication**, while using FirmWire to test the LTE RRC, GSM CC, and GSM SM protocols. We verify that these vulnerabilities are real and affect physical devices, demonstrating FirmWire's accuracy and strength as a baseband analysis platform.

Outline. The rest of this work is structured as follows: in Section II we briefly describe baseband firmware and motivate why we created FirmWire, Section III describes the design of FirmWire, Section IV provides implementation details of supporting the Samsung and MediaTek basebands, Section V evaluates FirmWire as a platform and its ability to find bugs, Section VI discusses additional research questions, Section VII highlights related work, and Section VIII concludes.

II. Motivation

Modern smartphones deploy a complex combination of hardware and software from many manufacturers. It is common to split the phone's functionality into an Application Processor (AP) and a BP. The AP is the user-facing processor. It runs the OS, such as Android or iOS, and all applications. The BP, which is not directly exposed to users, is responsible for all cellular network functionality.[1] This includes attaching and detaching from base stations and handling calls, Short Message Service (SMS) messages, and IP packets.

Due to the real-time demands of cellular protocols, BPs leverage an RTOS that is responsible for managing state machines, maintaining network timers, and processing complex message formats, such as ASN.1, CSN.1, and many other encapsulated protocols. Vendors commonly split processing of the complex cellular stack into independent *tasks* inside the RTOS, most often with a direct mapping between tasks and individual cellular protocols [46], [45], [64]. These tasks interact with each other using messages, which are sent and received via well-defined APIs [64], [21].

Understanding and auditing BP security is prohibitively difficult due to the tremendous complexity of modern cellular protocol stack implementations and a lack of baseband-specific testing platforms. We observe that recent attempts to tackle the challenges of automated baseband analysis suffer from limitations and do not offer the advantages of a full-scale dynamic baseband testbed. Such work typically falls into one of the following three categories:

1) **Dynamic on-device testing** tests physical UEs by sending cellular messages over-the-air and observing their behavior through oracles [40], [55], [17]. Such approaches offer only coarse-grained insight into the baseband's inner state. Even approaches for monitoring baseband diagnostics, such as SCAT [30], require a rooted device, which may be prevented by vendors.
2) **Emulation-based parser testing** extracts, rehosts, and tests individual parsers embedded in the cellular baseband [45], [22]. This approach not only requires extensive manual effort to support testing of new targets but also falls short in finding bugs beyond the initial function [45].
3) **Specification-driven static approaches** try to infer erroneous operations in the cellular specification [10] or uncover mismatches between baseband implementations and specifications [38]. Static approaches may be tailored towards specific protocols and operate unaware of the operating system context or state, yielding bugs not reproducible on real devices.

[1]Cellular basebands are not limited to smartphones. FirmWire's approach is applicable to other, more embedded targets, but for the purposes of this paper we only consider smartphone basebands.

III. Design of FirmWire

FirmWire expands upon previous baseband research by performing full-system emulation, scaling to many firmware images, and offering introspection. Figure 1 shows the components of the FirmWire platform. There are roughly two main parts: the vendor plugins (left) and the analysis core (right). A vendor plugin loads and parses raw firmware images to provide a layer of abstraction around the target for the analysis core. First, the vendor plugin extracts out the sections, CPU architecture, and load address, which are required by the analysis engine to begin the emulation process. Second, the vendor plugin uses signatures to locate core RTOS functions and data symbols across firmware revisions. The analysis engine links to these symbols to implement and expose the FirmWire interface, its common analysis API, to automated analyses (right block of Figure 1).

At the heart of FirmWire's analysis engine is a reconfigurable emulation core used to execute the firmware itself. To implement dynamic analysis, such as fuzzing, FirmWire provides a ModKit (lower right side) that allows for custom code to be injected into a running baseband. Module injection uses internal baseband RTOS APIs recovered by the vendor plugin and exposed by the FirmWire interface. The results of automated analyses (e.g., crash cases for fuzzing) can then be further processed by an analyst (rightmost side of Figure 1), for instance via over-the-air replication. To further aid the analyst, FirmWire offers rich debugging and introspection capabilities.

A. Vendor Plugins

Enabling FirmWire for different baseband vendors requires the creation of a vendor plugin. These plugins are created through one-time semi-automated effort, but scale to many firmware images outside of the initial development set. This is due to FirmWire APIs, which provide services for firmware loading, symbol discovery (PatternDB), System on Chip (SoC) selection and quirk handling, memory map extraction, and a built-in peripheral library. Vendors typically have a custom file format for delivering baseband firmware. These have unique headers and file extensions, making plugin selection easy and automatic from the first bytes of the firmware image.

Firmware Loader and Device Selection. When a plugin is selected, a firmware loader specific to a baseband vendor is executed. This loader carves the firmware file header into sections. From these headers, the high-level memory map and SoC version can be determined. Firmware loading does not require external tools or analyst intervention by design. The logic is completely integrated into FirmWire and executed on-demand without any image preparation required, meaning firmware directly from device updates can be used. This differs from previous work [38] that relies on external static analysis tools to preprocess images offline.

Baseband vendors typically provide multiple SoCs that are used by phone manufacturers across different devices. New SoCs are effectively hardware revisions, which change the memory layout and hardware peripherals, along with their locations. These unique attributes are consolidated by FirmWire into a single SoC definition file, with a simplified example shown in Listing 1.

```
from firmwire.peripherals import *

class VendorBaseSOC:
    common_peripherals = [
      SOCPeripheral(UARTPeripheral, base=0x84000000,
           size=0x1000)
    ]

class SOC123(VendorBaseSOC):
    name = "SOC123"

    # SoC specific peripherals
    peripherals = [
      SOCPeripheral(PMICPeripheral, base=0x80000000,
           size=0x100)
    ]

    # SoC specific attributes
    CHIP_ID = 0x01230000
    SOC_BASE = 0x82000000
    TIMER_BASE = SOC_BASE + 0x8000
```

Listing 1: Simplified SoC definition. `UARTPeripheral` and `PMICPeripheral` are pre-made peripherals either built in to FirmWire or specific to a vendor.

```
name: "BootTable",
pattern: [
  # Search for two stable 4-byte values
  "00008004 200c0000",
  # Alternate pattern for other images
  "00000004 ????0100"
],
required: true, # Fail boot if not found
# Process the found table and extract
post_lookup: parse_memory_table,
# Adjust found address
offset: -0x14,
# Make sure 4 byte aligned
align: 4
```

Listing 2: Example entry in PatternDB.

PatternDB. Since FirmWire is a platform for *binary* firmware, we must recover the location of important functions and data within each firmware image, even without symbol tables. For example, to know the currently running OS task, there could be a task ID stored somewhere in memory. Knowing this memory location would allow FirmWire to also know the running task and display this during execution. To address this requirement, FirmWire provides *PatternDB*, a configurable signature database. The most basic patterns contain a hexadecimal text string with optional wildcards. Wildcards are useful as they create patterns robust to minor changes, such as padding or slightly different instruction sequences, in different versions of firmware. Vendor plugins provide patterns, which are then processed over the bytes of a firmware image; successfully matched patterns are then added to FirmWire's symbol table. Such patterns are constructed by an analyst who has identified an address of interest using a static analysis tool. We provide a representative example in Listing 2 which is used for finding a data table crucial to loading and booting firmware. The exact syntax for pattern entries is described in Appendix A.

B. FirmWire Emulator

FirmWire's emulator is responsible for the execution of firmware, based on the information collected by the vendor plugin, i.e., the CPU configuration, expected memory regions,

their properties (read, write, execute), their contents (uninitialized or data bytes), found patterns, and peripheral locations. The emulator offers low-level access to baseband memory through the FIRMWIRE interface (middle of Figure 1).

Peripherals. One of the major challenges for emulating firmware is the missing interaction with physical hardware. Baseband firmware is no exception as it interacts with a diverse set of mostly undocumented peripherals, which must be implemented to prevent crashing or hanging during firmware execution. Fortunately, in many cases emulated peripheral memory responses can be reduced to a simple access pattern [25], [26], [19]. FIRMWIRE provides ready-made peripherals (CyclicBitPeripheral in Appendix Table IX) for these cases. If these are not sufficient, FIRMWIRE enables a semi-automated read-eval-print-loop (REPL) flow allowing analysts to repeatedly run the firmware with different peripheral behaviors until the boot (or other) process continues. This semi-automated human-in-the-loop approach is a one-time cost per peripheral.

FirmWire Interface. To abstract away from the low-level details of FIRMWIRE's emulation core, we leverage the vendor plugin's PatternDB to expose internal baseband functions with the outside world in the form of cross-platform APIs. The API includes verbs such as START, STOP, WRITE, READ, SNAPSHOT, SNAPSHOTRESTORE. Atop these low-level commands, we build FIRMWIRE's RTOS-level functions, including but not limited to LISTTASKS, INJECTTASK, and SENDMESSAGE. FIRMWIRE also provides a library of passive monitoring callbacks for baseband analysis. Examples include redirecting debug output of the modem to the analysis host, logging core activity of the RTOS scheduler, and logging internal messages.

C. Automated Analysis

Until now, we have talked about supporting new baseband platforms and controlling their execution at a low level. With the current foundation, we can begin testing protocols and baseband features at a high level.

Baseband ModKit. Building analyses may require changing baseband code to set up state and target certain protocols. As such, FIRMWIRE provides a firmware "ModKit" (Figure 2) which enables the development, compilation, and injection of data and code modules into the emulated baseband's memory. ModKit code is written in portable C and compiled to Executable and Linker Format (ELF) files using an off-the-shelf compiler matching the target's Application Binary Interface (ABI). During injection, modules can link against binary baseband functions using the help of FIRMWIRE's dynamic linker. This linker uses symbols from PatternDB to create a table of function pointers or data at runtime, similar to the dynamic loader for ELF files on Linux.

Symbols required for a module are indicated by a special pre-processor macro which defines a function prototype: `MODKIT_FUNCTION_SYMBOL(return_type, name, arg1, ...)`. FIRMWIRE will process these prototypes and emit `Write` commands to populate a table of function pointers. A module is injected by adding it to the RTOS' task structures, enabling the baseband itself to start the code.

Automated Protocol Testing. Using the ModKit, we developed a built-in module to enable coverage-guided fuzzing

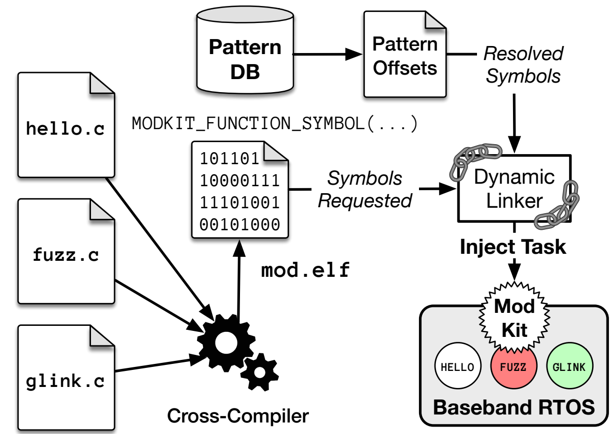


Fig. 2: The overall design of FIRMWIRE's portable ModKit. Modules are written in C and compiled using a baseband-specific cross-compiler matching the CPU architecture. These modules (ELF files) are dynamically linked against the PatternDB for portability between firmware and injected into the baseband RTOS to modify its functionality.

(discussed in more detail in Section IV-B). This combines built-in instrumentation of the emulation core and RTOS task injection. The injected task uses baseband messaging and task injection to fuzz cellular protocol implementations from within the baseband itself. This code must follow the rules of the baseband, including using the correct APIs. Normal baseband tasks communicate to each other with messages and our fuzzing tasks are no different. Multiple messages in sequence can be sent to any number of tasks, enabling complex state setup. By using the sanctioned interfaces for sending fuzzing data, we avoid entire classes of false positives due to uninitialized state. In contrast, BaseSpec [38] and BaseSAFE [45] may suffer from false positives, as they do not initialize the baseband through boot, but require the analyst to model the respective baseband state, and inject messages at non-task boundaries. Another benefit of messages injected directly using the baseband's RTOS APIs is that the message routing logic is handled by the baseband itself. FIRMWIRE effectively uses the same API that the baseband developers themselves use.

D. Introspection

Beyond automated testing, FIRMWIRE offers baseband introspection, allowing analysts to interactively explore the baseband to help prototype automated analyses. It also assists with root cause analysis for crashes generated by FIRMWIRE's automated analyses.

Function Hooking and Debugging. Building on PatternDB, FIRMWIRE is able to hook baseband APIs to record their arguments and return values. This is useful for understanding how the baseband operates when building automated analyses or performing root cause analysis. One of the most useful hooks is on internal baseband logging APIs. By hooking these APIs, we can liberate verbose baseband messaging. This is not only useful for bringing up new vendor plugins, but also for root cause analysis when debugging crashes. Beyond function

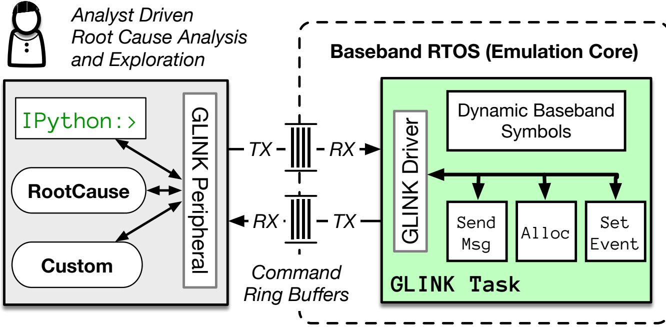


Fig. 3: An overview of the layout and communication between FIRMWIRE and the injected Guest Link (GLINK) RTOS task. An analyst can use the GLINK interface to manually send ad-hoc commands or execute one-shot analyses. These commands are effectively remote procedure calls serialized across a ring-buffer peripheral that the GLINK task can access.

hooking, we integrate FIRMWIRE with the GNU Debugger (GDB) to enable single stepping and memory exploration.

Guest Link (GLINK). To enable a more interactive exploration mode for FIRMWIRE, we created the Guest Link module (Figure 3). When not performing automated analysis, the GLINK module can be injected into an emulated baseband's task list. This task allows interactive, bidirectional Host to Guest communication via a custom hardware peripheral. GLINK's custom peripheral provides MMIO registers implementing a transmit and receive ring-buffer (i.e. a FIFO queue). This allows full-duplex information exchange between the baseband and FIRMWIRE APIs or a live analyst. An interactive shell or scripts running on the host can then inject messages, trigger events, and receive results. GLINK allows for asynchronous interaction with the emulated baseband without disrupting the regular operation of the firmware (meaning the target can be running). Extending GLINK only requires adding a single opcode, command handler, and `struct` definition shared between FIRMWIRE and the injected task's code.

IV. IMPLEMENTATION

To demonstrate FIRMWIRE, we created vendor plugins for Samsung and MediaTek basebands. We chose Samsung and MediaTek because both vendors provide chipsets for millions of devices worldwide, accounting for more than 40% of the worldwide shipped BPs in 2020 [15]. No public documentation exists for the hardware and firmware of these platforms, but they have previously been the subject of some reverse engineering and exploitation by the industry [6], [21], [22], [48] and, more recently, academic research [40], [38], [55], [45].

A. Vendor Plugins

Within each vendor plugin, we create a loader, populate PatternDB, and create SoC definitions. Plugins are written in Python for rapid prototyping and integrate with the Platform for Architecture-Neutral Dynamic Analysis (PANDA) [16], a QEMU-based emulator [8]. We extended PANDA to support the needs of our baseband targets, which included adding new instructions for MediaTek. To integrate these components, we extended the avatar[2] framework [49], which acts as middleware for FIRMWIRE, enabling Python-based peripherals.

Vendor	Phone Model	Chipset	#SLoC
Samsung	Galaxy S7/S7 Edge	S335AP	25
	Galaxy S8/S8+	S355AP	29
	Galaxy S9	S360AP	33
	Galaxy S10/S10e	S5000AP	25
MediaTek	Galaxy A10s	MT6762	14
	Galaxy A41	MT6768	12

TABLE I: The different smartphones and corresponding SoCs that are supported by FIRMWIRE.

For our implementation, we select 9 different popular phone models from the Galaxy line which use 6 different chipsets (Table I). While recent work focused on the older MediaTek firmware for ARM [45], [38], our selection includes the recent MediaTek chipsets based on the MIPS ISA, demonstrating alongside Samsung's ARM-based Shannon chipsets the architecture independence of our approach.

Samsung's Baseband. Samsung baseband processors, until the S5123AP, are based on the Cortex-R microprocessor family. They run a proprietary real-time operating system (ShannonOS) and implement several cellular stacks, including GSM, UMTS, LTE, and 5G NR on recent modems. Interaction with the main Application Processor (AP) is carried out via DMA to a custom kernel driver and inter-processor interrupts. After writing a loader for Samsung's firmware images, which follows a known format [21] across all firmware we examined, we developed 18 PatternDB entries (see Appendix Table VII) to smooth over the address and data variations between firmware, enabling stable operating system introspection of running tasks, messaging APIs, and memory maps. We next performed the semi-automated peripheral implementation process as described in Section III-B, until our baseband introspection indicated that tasks were coming online and no exceptions were raised during baseband execution. For FIRMWIRE to boot the Samsung baseband, we provided 17 peripheral definitions total, with some being SoC specific (see Appendix Table IX).

After implementation, our Samsung vendor plugin is capable of starting nearly all RTOS tasks, though some raise exceptions. These were disabled using the RTOS API if and only if they were deemed to not be critical for protocol testing. To our knowledge, no previous work has emulated the Samsung baseband RTOS with enough fidelity to boot tasks to this level. In total we now support 4 Samsung baseband SoCs across 7 phone models (Table I), allowing us to boot nearly 200 different firmware images across 5 years of devices (Table VI). For further details, see Section V-E where we discuss the bring-up of the S335AP SoC along with additional firmware images.

MediaTek's Baseband. Recent MediaTek BPs are based on a MIPS (interAptiv) processor, with several custom DSPs and other baseband-specific peripherals. Previous work [38], [45] explored legacy versions of this baseband based on ARM. Our emulation core, PANDA, did not support this specific architecture out of the box, hence we needed to add support for missing instructions, primarily those for MIPS16e2, a compressed instruction set similar to ARM Thumb. This support

consisted of adding new instruction decoding and Tiny Code (TCG) lifting.[2] For MediaTek's firmware format we wrote a loader to extract the relevant code and data regions. MediaTek firmware luckily contained a symbol table, unlike Samsung, which greatly simplified the creation of PatternDB entries. Only 9 patterns were needed to scale up to multiple firmware images (see Appendix Table VIII). Like Samsung, we repeated the semi-automated peripheral implementation process for MediaTek's peripherals. In total only 12 peripherals, many of them just placeholders, were required to support 2 SoC platforms (see Appendix Table IX).

One of the most complex peripherals that required modeling was the communication between the MediaTek BP and AP. System configuration information such as memory regions is provided by the AP, so this interface is required for boot. Support included instantiating ring buffers and messaging structures (implicitly documented by MediaTek's Linux kernel source code). This interface is also used at runtime, primarily to make requests to userspace programs; we implemented support for running one such program (for storage access) alongside FIRMWIRE, replacing the kernel interface with a FIFO. The final obstacle in the boot process involves the BP waiting for other hardware threads, which PANDA does not support; we use PatternDB to find and bypass the code in question and adjust all task affinities to run on a single core. As with Samsung, our MediaTek implementation in FIRMWIRE is capable of running nearly all RTOS tasks without failures; we only needed to disable the low-level RF and hardware-related tasks. To the best of our knowledge, this is the first public work to emulate the MediaTek baseband RTOS to this level of fidelity. In total we now support 2 MediaTek baseband SoCs across 2 phone models (Table I), allowing us to boot 9 different firmware images across nearly 2 years of devices (Table VI).

B. Automated Analysis

With 213 bootable baseband images now running under FIRMWIRE, we turn to security analysis. To do so, we chose coverage-guided fuzzing combined with semi-automated root cause analysis (Figure 4). This approach has only been used on single functions in previous research [45], while we perform it on a fully running baseband. While some argue that dynamic analysis of basebands is limited or even ineffective due to the statefulness of cellular protocols [38], FIRMWIRE counters these objections by emulating the *entire* baseband, achieving a fine-grained initial state usable for fuzzing. By injecting messages to the baseband in this initialized state, new states can be reached to also be used for testing.

FUZZ Task. We created the FUZZ module using FIRMWIRE's ModKit. This drives coverage-guided fuzz testing from within the baseband's memory. This task requests test cases from the host via guest-to-host hypercalls [52], enables coverage collection and embeds them into messages sent to target cellular protocol tasks. Hypercalls are implemented as special instructions that, when translated by the emulator, trap out of the guest OS to call a host helper function of our choosing. To capture crashes the FIRMWIRE vendor plugin configuration is used to determine the relevant events, such as CPU exceptions

[2] This also included extending Ghidra's MIPS SLEIGH lifter to emit PCode for these instructions.

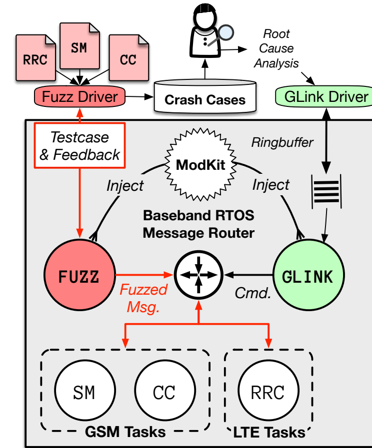


Fig. 4: Exemplary application of FIRMWIRE for security analysis. Special FUZZ tasks are injected into the running baseband for coverage-guided fuzzing. The resulting crashes are then subject to root cause analysis, which is aided by the GLINK task.

or deliberate baseband `asserts`. Our anomaly detection is not limited to crashes. Additional conditions can be added to match the specific fuzzing campaign if needed. When a fault occurs, it is automatically saved for analyst review and can be replayed to obtain crash context and internal baseband log messages. Our FUZZ Task can be modified using a few lines of C to target arbitrary tasks and messages as shown in Table II, giving the analyst freedom to evaluate arbitrary parts of the BP. For input generation, we build on top of the state-of-the-art fuzzer, AFL++ [20], which runs on the host. Once a FUZZ task is written, it is injected into the baseband using our ModKit (Figure 2).

V. EVALUATION

We evaluate FIRMWIRE by posing three questions:

- **Q1** Can a dynamic emulation platform aid security testing for baseband firmware?
- **Q2** Is FIRMWIRE's approach scalable?
- **Q3** Are vulnerabilities discovered with FIRMWIRE real bugs affecting physical phones?

For **Q1** we wrote fuzzing tasks for multiple cellular protocols (Section V-A), evaluate the achieved fuzzing performance (Section V-B), and compare the results with the state of the art (Section V-C). Using these fuzzers, we found in total 7 vulnerabilities (Section V-D) of which 4 were previously undiscovered and with 1 acting as our ground truth. For **Q2**, we showcase an empirical measurement of effort required to integrate a new hardware platform and firmware images to the FIRMWIRE platform (Section V-E). We further demonstrate FIRMWIRE's scalability by dynamically replaying crashing inputs for 229 firmware images across 9 different device models (Section V-F). Lastly, for **Q3** we reproduce 6 vulnerabilities over-the-air, indicating the real-world impact of vulnerabilities found in our emulator (Section V-G).

	FUZZ Task	SLoC	#Setup Messages
MediaTek	RRC (4G)	162	0
Samsung	RRC (4G)	53	0
	SM (2G)	51	1
	CC (2G)	46	1

TABLE II: Fuzzing tasks injected into the basebands.

Chipset	Phone Model	Release	FW Image
Samsung S355AP	Galaxy S8	Sep'17	**G950FXXU1AQI7**
Samsung S5000AP	Galaxy S10	Feb'19	G973FXXU1ASBA
	Galaxy S10	Mar'21	**G973FXXU9FUCD**
	Galaxy S10e	Nov'19	G970FXXU3BSKL
	Galaxy S10e	Mar'21	**G970FXXU9FUCD**
MediaTek MT6768	Galaxy A41	May'20	**A415FXXU1ATE1**
	Galaxy A41	Jan'21	**A415FXXU1BUA1**

TABLE III: Target images for fuzzing experiments. Images with bold name were used for the performance evaluation.

A. Security Analysis: Fuzz Testing

Although a full-fledged baseband emulation platform offers many dynamic analysis capabilities, we chose fuzz testing as the main application for FIRMWIRE. In particular, we want to automatically explore and test complex radio message handling within the baseband and use the introspection capabilities provided by FIRMWIRE to identify the root causes of crashes more precisely than previous work [40].

Attacker Model. For identifying which parts of the modem firmware to fuzz, we deployed a simple, yet realistic threat model. Our attacker model focuses on discovering proximate attacks involving a UE baseband receiving data from a nearby rogue base station. We assume the attacker controls the rogue base station and is able to send arbitrarily crafted control plane radio payloads to be processed by the UE. However, we do not assume that authentication between base station and UE was carried out to guarantee that attacks can be carried out on real devices. Hence, we focus our testing on the pre-authentication attack surface of LTE and protocols with broken authentication (GSM). UEs today may roam to GSM networks if they are the only ones available or through a downgrade attack by rogue base stations [61].

Fuzzing Targets. After carefully considering the attacker model, we wrote fuzzing tasks for 3 parts of the cellular protocol stack. Specifically, we tested LTE Radio Resource Control (RRC) [3], GSM Session Management (SM) [4], and GSM Call Control (CC) [4] protocols.

For each of these protocols, we create a fuzzing module that uses the FUZZ task as a base (Table II). Once written, the tasks are injected into each baseband. These tasks will be scheduled by the RTOS and carry out the necessary target initialization by injecting messages using the corresponding RTOS API. They then activate the AFL++ forkserver and begin distributing fuzzing inputs to target tasks (e.g., RRC, SM, CC) using the RTOS messaging APIs recovered by PatternDB. The inputs are encoded OTA messages. This means UPER encoded ASN.1 for RRC and CSN.1 encoded Non-Access Stratum (NAS) packets for SM and CC.

Image Selection. For our fuzzing experiments, we use the chipset, models, and firmware as shown in Table III. We chose these chipsets based on their use in Samsung's flagship (Galaxy S10 and earlier) and budget (A41) models. At the time of writing, these devices are widely available and well-supported with regular security updates. The S8 is no longer receiving security updates and was instead used for our ground truth testing. Note that we are *not* limited to fuzzing the 7 images in Table III and support significantly more firmware for fuzzing (see Section V-F). Unlike previous work, we purposefully chose the A41 model as it has a recent MediaTek chipset, which uses the MIPS instruction set (rather than ARM) to demonstrate the flexibility of FirmWire's approach across different ISAs.

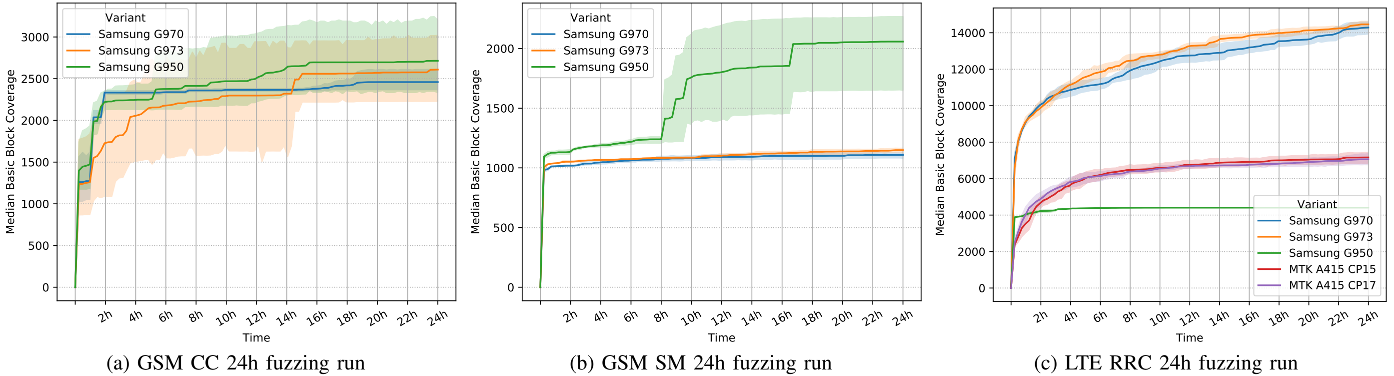
B. Fuzzer Performance

Apart from discovering bugs, we use fuzzing to answer two additional questions in this section: (1) Can our FUZZ tasks trigger complex behavior and reach deep code paths in the emulated baseband and (2) How do protocol implementations across baseband versions and vendors compare in complexity?

Setup. To conduct our evaluation, we used a server with 2 Intel Xeon E5-2630v4 @ 2.20GHz CPUs (20 physical and 40 logical cores) and 128GB of RAM running Ubuntu 16.04 LTS and drive FIRMWIRE with AFL++ v3.13a as a fuzzer. A single fuzzing instance uses less than 512MB of RAM, making the number of cores available the limiting factor. To allow multiple runs in parallel, we set the CPU affinity for each of the stochastic tests to a single core to limit scheduler noise. We run each FUZZ task for 24 hours, each in 5 independent runs. For the three selected Samsung images, we fuzz LTE RRC, GSM CC, as well as GSM SM. Additionally, we fuzz LTE RRC for MTK to be able to compare different implementations. During fuzzing, we use AFL's persistent mode with 1000 iterations. This means the fuzzer will perform 1k executions in the booted system before it resets the emulator. From previous tests, we found this to be a good trade-off between mutating baseband state and hard-to-replay test cases while minimizing slow resets.

Reached Basic Blocks. After all runs were completed, we re-ran the generated test cases and collected the unique translated blocks using FIRMWIRE's coverage instrumentation. We then mapped these QEMU blocks to the corresponding basic blocks using the Ghidra reverse engineering framework. This way we get the correct number of basic blocks for each test case, despite hash collisions in AFL++'s coverage map and inconsistencies caused by QEMU's translated blocks. The results of this experiment are visualized in Figure 5.

Our results show that while fuzzing RRC, we exercise a far greater amount of basic blocks (functionality) compared to other cases. For example, when fuzzing the Shannon RRC implementation our fuzzer exercises more than 10,000 unique basic blocks, while discovering less than 4,000 basic blocks for GSM CC and SM. We also observe that the curves of discovered basic blocks over time for the same FUZZ test behave similarly for different firmware and hardware platforms. The two exceptions are the fuzzing results for the G950 (green)



(a) GSM CC 24h fuzzing run

(b) GSM SM 24h fuzzing run

(c) LTE RRC 24h fuzzing run

Fig. 5: Discovered basic blocks over time for fuzzing campaigns across firmware images (median and 95th percentile). Each subfigure depicts one fuzzing target with data collected over 5 independent 24-hour runs for each selected firmware image.

SM and RRC fuzzing runs. For GSM SM, more basic blocks are discovered on the G950 model compared to the G970 and G973. For RRC fuzzing, the G950's block count caps at around $5,000$, unlike the other two models. As we show in Section V-F, this is in line with our fuzzing results. More specifically, we discover crashes in GSM CC for all tested firmware images, while only uncovering a bug for GSM SM in G950 firmware (which has the most code coverage). Similarly, our fuzzers found RRC-related crashes in all images *except* the G950 one (which has the least coverage). This behavior could indicate a significant re-architecture of the targeted code or changes to existing branch conditions that are difficult to bypass with simple mutation-based fuzzing.

Per-Task Coverage. The number of reached basic blocks are collected over the full modem operation including inter-task interactions and low-level OS execution. Hence, to further investigate the fuzzing efficacy, we also set out to measure the achieved per-task coverage. As modem images are distributed in binary-only form, we compensate for the lack of source code or binary labeling ground truth by using the proxies described in the Appendix when associating basic blocks of the firmware with specific tasks. We show the number of reachable basic blocks per task (according to the proxies) and the percentage of reached blocks over the fuzzing experiments in Table IV.

We first observe that the number of reachable blocks for RRC is significantly higher than for CC and SM. This maps directly to the complexity of the corresponding standards; the RRC specification spans over 1100 pages [3], while the GSM layer-3 specifications describing SM and MM contain around 800 pages [5].

Second, we see that the number of covered blocks is surprisingly low, ranging from only 1%-3.5% in the case of RRC to 14%-20% for CC. We believe this is due to two reasons. First, to reflect our attacker model, our injected FUZZ tasks target only a subset of the internal messages which would be normally processed by the tasks under test. As such, large parts of the tasks are made unavailable to the fuzzer from the beginning. Second, cellular protocols are inherently stateful with a large number of internal state machines. While FIRMWIRE boots the baseband into an operable initial state, systematically exploring complex state spaces in an efficient manner poses an open challenge to coverage-guided fuzzing approaches [7] and is a problem outside the scope of this paper.

Despite the seemingly low coverage, FIRMWIRE was able to uncover several critical vulnerabilities as we will show in Section V-D. Hence, we believe that virtualized testing of baseband firmware is a viable strategy with room for future work to improve generating inputs or new states that exercise deeper paths in the firmware.

C. *Comparison with the State of the Art*

The amount of prior art on fuzz testing cellular baseband implementations is limited. To this day, the most comprehensive study focusing on fuzz testing UEs is LTEFuzz [40], which directly tests physical devices in a black-box manner. Unfortunately, the source code is not openly available and typical test durations are not reported in the paper, which makes an experimental comparison to LTEFuzz non-trivial.

The other most prominent work on UE fuzzing is BaseSAFE [45], which is openly available. This approach selectively rehosts and fuzz tests message parsers in baseband firmware. In contrast to FIRMWIRE, creating a fuzzing harness in BaseSAFE requires manually crafting an initial state which is then loaded into the Unicorn emulator for fuzzing [56]. To compare to FIRMWIRE, we created harnesses for a selected firmware image in two different ways.[3] Firstly, as proof of concept, we manually created a harness for the CC message parser and verified with the BaseSAFE developers that we correctly followed their approach. We did not manually implement harnesses for SM and RRC due to the required manual

	CC		SM		RRC	
	BB_{cov}	BB_{max}	BB_{cov}	BB_{max}	BB_{cov}	BB_{max}
G950	17.28%	9067	8.44%	6686	3.54%	54531
G970	14.01%	9161	4.13%	5785	2.82%	89584
G973	19.71%	8555	5.35%	5552	2.85%	90497
A415 CP15		-		-	0.99%	68056
A415 CP17		-		-	0.99%	68100

TABLE IV: Per-task coverage for fuzzed images.

[3]Note that we could only use the ARM based Samsung images, as the MIPS dialect used by MediaTek is not supported by Unicorn.

	CC		SM		RRC	
	BB_{cov}	BB_{max}	BB_{cov}	BB_{max}	BB_{cov}	BB_{max}
Manual	15.55%	8555	-	-	-	-
Automated	25.30%	8555	7.49%	5552	2.80%	90497

TABLE V: Per-task coverage with BaseSAFE fuzzers for G973FXXU9FUCD.

effort (multiple days per firmware, per fuzzing target). Instead, we wrote wrappers to automatically import an initialized memory snapshot from FIRMWIRE into BaseSAFE. Using these wrappers, we created snapshots with the injected fuzzing tasks and provided hooks to correctly retrieve fuzzing input via BaseSAFE.

We then ran five 24-hour experiments for each of these harnesses on the same machine used for the fuzzing performance evaluation, and report the resulting per-task coverage in Table V. It is important to note that although BaseSAFE supports persistent mode, we could not use it for these experiments due to incompatible programming models for the fuzzers. In comparison to FIRMWIRE, the manual BaseSAFE harness for CC achieves less coverage (19.71% vs 15.55%), which is in line with our expectations. In contrast, the automatically generated harnesses exercise more (CC & SM) or an equivalent amount of code (RRC) in the targeted tasks. In this limited scenario, it appears that fuzzing without full-system emulation mode achieves higher code coverage due to less CPU overhead. However, this comes at the cost of no longer having an accurate operating system model as task scheduling, timing, and messaging semantics no longer exist. Additionally, setting up an initial state corresponding to our automatically generated harnesses would be non-trivial without the accurate snapshot already provided by FIRMWIRE's system emulation. To conclude, we believe that FIRMWIRE is not only useful as a standalone framework but can also integrate well with other state-of-the-art approaches.

D. Discovered Vulnerabilities

Our fuzzing campaigns resulted in a variety of crashes across the different baseband versions. After de-duplication and root cause analysis, we attribute these crashes to 7 distinct vulnerabilities (4 RRC, 1 SM, and 2 CC).

LTE RRC #1: RRC Reconfiguration. This crash case occurs when providing malformed *RRCConnectionReconfiguration-r8-IEs* within a *RRCConnectionReconfiguration* message. Although the LTE RRC specification [3] specifies that this message is conditionally accepted before the over-the-air security establishment, the message has to be decoded regardless of the security status in order to identify it. Thus, the decoder is always called and this crash can be triggered at any time once a UE is idle on a cell. Interestingly, the decoder infers an invalid length but does not crash right away. Instead, the decoded message is re-encoded onto a fixed-size stack buffer. Due to a wrongly inferred length in the previously decoded IE, this buffer is exceeded, causing corruption of the saved instruction pointer with attacker-controlled data, which can result in remote code execution. This example demonstrates the advantages of FIRMWIRE in comparison to approaches only targeting specific decoders (e.g., [45], [38]), as the vulnerability is not in the decoder itself, but in subsequent re-encoding.

LTE RRC #2: RadioResourceConfigDedicated IE. This crash occurs when providing a *RadioResourceConfigDedicated* IE containing a malformed *Mac-MainConfig* field. This IE can be included in various RRC messages, including *RRCConnectionReconfiguration*. Similar to LTE RRC #1, the corruption occurs during later re-encoding, but instead of crashing with a `PREFETCH ABORT` (ARM) due to a corrupted instruction pointer, the baseband issues a `PAL_MEM_GUARD_CORRUPTION` exception. Upon closer investigation, we identified that in this crash case a heap buffer is overflowed, which causes the baseband to raise the exception upon freeing the corresponding chunk. The root cause is when freeing an allocation, the modem heap allocator modem checks for heap sentinels (`0xaaaaaaaa`), which no longer match, leading to the fatal assertion. This serves to crash the baseband but offers no security benefit as these guards are hard-coded and can be replaced, allowing an attacker to corrupt adjacent heap objects and metadata without crashing during free.

LTE RRC #3: RRC Reconfiguration. As with LTE RRC#1, this vulnerability is triggered by a *RRCConnectionReconfiguration* message but requires a malformed *RRCConnectionReconfiguration-v1250-IEs* instead. Despite using different IEs, the baseband's behavior upon receiving such a malformed message is identical to LTE RRC #1: A `PREFETCH_ABORT` is issued due to a saved instruction pointer on the being corrupted during re-encoding. While the crash behavior appears identical, we used FIRMWIRE's introspection capabilities to verify that these are indeed two distinct vulnerabilities, as the overflown stack buffers reside in different functions.

LTE RRC #4: MCCH Double Free. This issue was found by our LTE RRC fuzzer in the Galaxy A41 firmware images, which use the MediaTek baseband. Most of this code has previously been extensively fuzzed by previous work [45]. However, almost immediately after beginning our campaign, FIRMWIRE reported inputs that caused assertion failures (leading to a reboot of the BP). These are caused by buffers being freed twice on the code path after a MCCH message has been received and the ASN.1 decoder has reported an error while decoding it. Again, this is detected by the baseband's own heap allocator checks, and would not be visible if we had limited our fuzzing to the ASN.1 decoder code.

SM #1: GPRS Session Management PDP Activation. This particular vulnerability is special because it was previously known and we specifically searched for it. Using research from [6], we set out to replicate the findings automatically. All work performed by this previous research was entirely manual static analysis and over-the-air testing, which does not scale. To serve as ground truth for FIRMWIRE's emulation fidelity and accuracy, we identified the target protocol task and wrote a fuzzer targeting it. Our fuzzer successfully generated a crashing input triggering the bug embedded within the `PDP CONTEXT ACTIVATION` [4] message handler. The root cause of the crash is missing validation of a length field when decoding a Type-Length-Value (TLV) Information Element (IE), leading to a classic stack-based buffer overflow, enabling remote code execution due to return address corruption. The target raised a `PREFETCH ABORT` exception which FIRMWIRE

used to flag the test case. Using the introspection offered by our platform, especially baseband logs and debugging support, we could pinpoint the exact crash condition to invalid processing of the Protocol Configuration Options (PCO) [4] IE.

CC #1: Call Setup Heap Overflow. This vulnerability in a GSM stack implementation was discovered by our CC fuzzer during a large-scale fuzzing campaign, lasting 25 cumulative days of CPU time across 30 instances, totaling 283 million test cases. The vulnerability is triggered by the `CALL SETUP` packet [4], which is sent when the base station establishes a Mobile Terminated (MT) call. It is responsible for exchanging metadata, including the remote bearer's capabilities, including available voice codecs.

The crash occurs during the processing of the bearer capability IE (10.5.4.5). It should never exceed 16 bytes, yet the implementation trusts the OTA length field of the TLV during a `memcpy`, leading to a heap overflow. Using GDB, we could confirm that in this case, the data overwriting the adjacent chunks is directly controlled by the attacker, without prior ASN.1 encoding or decoding as in the case of LTE RRC #1 & #3.

CC #2: ASN.1 Decode Error. The second CC corruption occurs earlier during setup packet decoding and is the result of an invalid decoding procedure for the ASN.1 encoded *ss-Code* information element for supplementary services [5]. Supplementary services provide additional functionality related to voice calls, such as call forwarding [2] and call barring [1]. Signaling messages related to supplementary services are usually delivered during voice call setup, including the status notification of certain supplementary services. To deliver the notification, the *opCode* IE is set to *notifySS* and *ss-Code* IE is set to the service-specific values.

When the received IE is malformed, instead of raising an error and aborting, the ASN.1 decoder attempts to decode the IE at a later offset in the received data, while incrementing the expected length. This results in writing more data than expected to a stack buffer, resulting in corruption of the saved program counter, ultimately leading to another `PREFETCH ABORT`.

Coordinated Disclosure. With the exception of SM #1, which was previously known and used for ground truth testing, we reported all found vulnerabilities to Samsung. We reported the presented vulnerabilities in the LTE stack in 2021 and the ones in the GSM CC implementation in 2020. Samsung determined that CC #1 was patched in early 2020 independent from our report (bug collision), and considered LTE RRC #1 as a duplicate due to its similarity to our other findings (same bug, multiple paths).

Samsung classified all other found vulnerabilities **as previously unknown over-the-air vulnerabilities affecting multiple devices**. LTE RRC #2, #3, and CC #2 were assigned critical severity scores, whereas LTE RRC #4 was assigned high severity. Samsung allocated SVE-2021-22079 (CVE-2021-25479), SVE-2021-22051 (CVE-2021-25478), SVE-2020-18098 (CVE-2020-25279), and SVE-2021-22199 (CVE-2021-25477) respectively for our findings, and deployed patches in the fall of 2020 and 2021.

E. *Extending* FIRMWIRE

To give the reader insight into the extensibility of FIRMWIRE, we conducted an empirical measurement of the required time and effort to add a new hardware platform and additional firmware to the framework. In particular, we added support for different firmware targeting Samsung S7 and S7 Edge basebands while recording the time and taking notes. While this is not an ideal experiment, we believe that this can provide intuition about both the automation offered by FIRMWIRE and the manual effort required to create or modify vendor plugins.

Additional Hardware Platforms. We started by choosing a single G930 firmware image from mid-2018 and tried to load our emulator. FIRMWIRE detected the hardware platform as a S335AP, which was not supported. We copied the nearest SoC definition, the S355AP, as a template and registered the S335AP in the vendor plugin (26 lines of Python). Booting the image caused a bootloader crash mentioning an invalid chip ID. We adjusted the `CHIP_ID` to `0x03350000`. Booting again made it further, but now the image hung without log output. This was expected as the S355AP template contains peripheral base addresses that may have changed. Using a dedicated debug flag for displaying all peripheral accesses, we discovered an infinite loop on a read. We recognized the access pattern as that of an already made peripheral and adjusted the base address accordingly (1 line change). Booting again, the firmware executes past the BOOT image and runs into the MAIN image where another hang occurred while enabling system clocks. Using debug and raw assembly display modes of FIRMWIRE, we noticed a while loop on a peripheral field access. To bypass this, we created a cyclic bit pattern of `0xfffff` [19], which passed the check and continued the boot significantly further (2 lines of Python). Next try, the image hit a fatal assert in `hw_ClkFindSysClkCofigInfoIndex`, which is related to clock configuration. To fix this, we extended a PatternDB entry (1 line of code) to find the code and pass the check for the S335AP (the current pattern only worked for the S355AP). Next try, we observed another fatal error of "check code sync between CPU and DSP" and adjusted the S335AP's reused DSP peripheral (1 line of code) to use the correct codes (which were displayed in the assert). For the next four boots, the firmware hung during the L1C, InitPacketHandler, PacketHandler, and SIM tasks. We disabled these one-by-one (4 lines of code) as they use unmodeled peripherals (radio hardware, chip-to-chip communication, and I2C respectively). Finally, the firmware booted without crashes or hangs. In total from start to finish, this process took less than two hours for one person acquainted with the framework.

Additional Firmware. Although the above approach already allows a variety of S7 and S7 EDGE firmware to run in FIRMWIRE, some images, particularly those created before 2018, did not run in the emulator. Hence, we took an additional firmware from 2016 and added support for it. While the SoC was selected correctly and FIRMWIRE attempted to load the firmware, the task table could not be resolved correctly. This was due to the modem internal task table structure using a different layout compared to the newer image. However, the required layout was already integrated into FIRMWIRE, and instructing the framework to use it for older S7 images required 2 lines of code. Afterward, the emulator failed to boot as

it could not recognize the required `log_printf` pattern, which could be resolved by adjusting the existing pattern via additional wildcards (1 line of code). Then, the firmware booted up to the synchronization check between CPU and DSP. As it turns out, different firmware revisions use different constant values when communicating with the very same DSP, and we resolved this issue by parsing the log message and using the adjusted code for firmware images before June 2018 (2 lines of code). Now, the firmware booted but fuzz testing failed, as `pal_MemAlloc` was not correctly resolved. This was addressed by adding an additional pattern for this image (1 line of code) resulting in the firmware booting. In total, this additional firmware required 3 hours and was carried out by a different analyst, who was not involved in creating platform support for the S335AP.

F. Large-scale Vulnerability Analysis

The previous sections demonstrate FIRMWIRE's targeted bug finding capabilities and extensibility but provide few insights about the actual scalability when applied to other baseband firmware images beyond the ones selected for fuzz testing. Hence, we conducted a large-scale (**Q2**) study where we ran the crash cases found by FIRMWIRE against firmware images outside our initial test set. More specifically, we tested against a representative data set spanning a wide variety of firmware for whose chipset we created vendor plugins.

Dataset collection & emulation efficacy. We downloaded all available firmware updates targeting a single region for 9 different mobile phone models using a public service [59]. These updates are not exclusively for the BP but are bundled together with updates to the AP. As a result, not all update files provide new baseband firmware images. In total, we downloaded 360 firmware updates and could obtain 229 unique baseband images of which we could successfully boot **213** using FIRMWIRE. The other images failed to boot due to errors in emulation fidelity, resulting, for instance, in hangs while waiting for the expiry of specific timers or infinite loops when checking for specific return values from peripherals. While these errors could be fixed, the purpose of this experiment is to demonstrate on how many images FIRMWIRE *does* work without additional manual intervention. Table VI details the distribution of the collected images over the different phone models.

Results. For each image booting up in our emulator, we inject the FUZZ task that triggered the specific crash and provide it with the crashing test case. We then monitor the behavior of the emulated firmware and log whether the baseband crashed.

Phone Model	Earliest	Latest	#Updates	#Images	#Booting
Galaxy S7	Mar'16	Nov'20	37	24	21
Galaxy S7e	Mar'16	Nov'20	52	27	24
Galaxy S8	Apr'17	Apr'21	47	37	34
Galaxy S8+	Apr'17	Apr'21	17	15	15
Galaxy S9	Mar'18	Oct'21	101	40	36
Galaxy S10	Feb'19	Oct'21	40	31	30
Galaxy S10e	Feb'19	Oct'21	40	31	29
Galaxy A41	Jun'20	Aug'21	13	12	12
Galaxy A10s	Dec'19	Sep'21	13	12	12
Total:	Mar'16	Oct'21	360	229	213

TABLE VI: Collected Firmware Updates and Images.

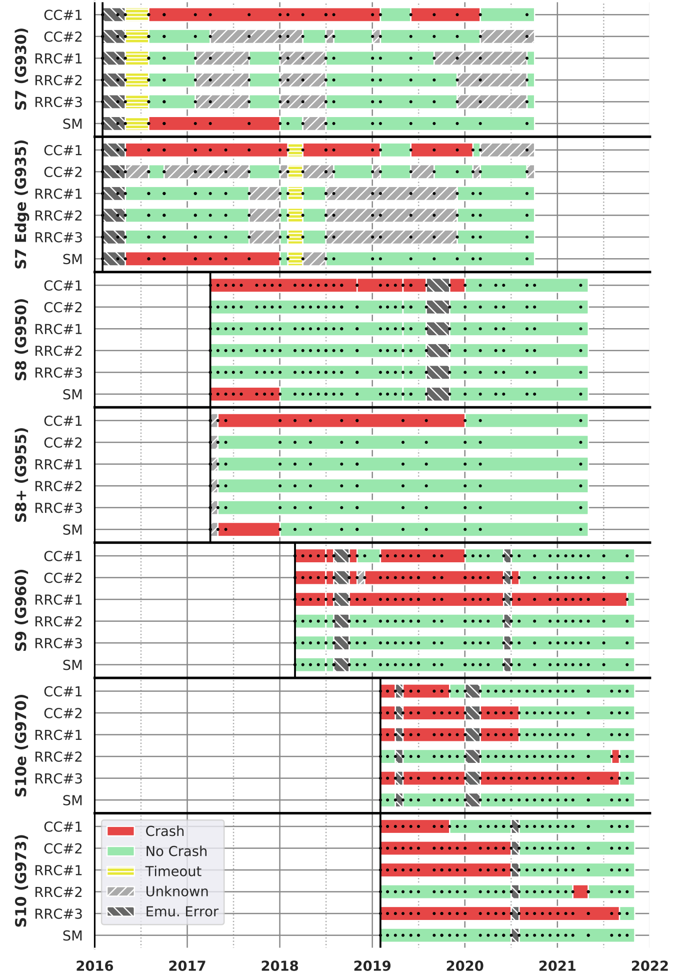


Fig. 6: Large scale testing of discovered Shannon baseband crashes over time, per phone model and firmware image. Each black dot is an image tested, with state interpolated between. "Crash" indicates that the image crashed when receiving the input. "No crash" means the image did not crash when receiving the input. "Timeouts" occur when the emulator could not retrieve and process the input in time. "Emulation Error" means FIRMWIRE was not able to boot the firmware. "Unknown" indicates other types of errors.

We showcase the results of our longitudinal study in Figure 6. We omit the results for LTE RRC #4, as this vulnerability only affects MediaTek chipsets, and all tested versions prior to August 2021 turned out to be vulnerable. The more recent firmware (only one per chipset at the time of writing) includes patches following the coordinated disclosure process with Samsung.

Analyzing the other results of the longitudinal study, we make several observations. First, although FIRMWIRE successfully boots the majority of tested images, both unknown and emulation errors occur for a variety of images, at different points in time. This is especially true for S7 and S7 Edge images, which is not surprising. As described in the last section, only two images were analyzed when adding support for these models to FIRMWIRE. We believe that in most

cases, the emulation errors are caused by subtle changes in task interfaces, which require additional PatternDB updates. Furthermore, single modem images are not crashing when supplied with specific crashing inputs, even though directly before and after released images do crash on the same input. This is likely caused by additional inaccuracies in the emulation or code churn from the baseband vendor affecting the crashing symptoms. The absence of a crash in the emulator does not necessarily imply the absence of the vulnerability.

Our second observation is that while most of our found vulnerabilities affect at least two or more chipsets (SM #1, CC #2, RRC #2 & #3), only one vulnerability seems to apply to all tested models (CC #1). We hence conclude that most vulnerabilities found in FIRMWIRE's emulator are likely to be present in other firmware images targeting similar, but not necessarily identical, chipsets.

Third, we observe that phones with the same chipset follow similar update schemes, and vulnerabilities are usually patched for multiple images simultaneously. Interestingly, our test case for the RRC #1 vulnerability does not lead to crashes on the Galaxy S10/S10e firmware images since mid-2020. Yet it continued to crash on recent Galaxy S9 firmware until the October 2021 update. We speculate that the baseband code base is fragmented across different chipset implementations, which could make patch propagation an error-prone task.

Additionally, we highlight that single fixes can patch multiple vulnerabilities at the same time. Specifically, one update in August 2020 fixed CC #2, and LTE RRC #1 for images using the S5000AP chipset. Although these crashes can be triggered via different cellular messages for distinct protocols and code paths, we believe they are all caused by improper validation of ASN.1 decoding output, regardless of whether the crash occurs immediately or at a later stage during execution.

At the time of writing, all tested vulnerabilities are fixed on the most recent firmware versions due to our disclosure with Samsung. FIRMWIRE not only allows introspection into single baseband firmware images but also valuable insights to development and patching cycles over time (**Q2**).

G. Over-the-air Reproduction

To prove the applicability of the emulated fuzzing results (**Q3**), we replay the found crashes over-the-air against a select set of devices. For over-the-air verification, we used Ettus Research's USRP B210 and N210, lab-grade off-the-shelf Software-Defined Radios (SDRs) for radio frontend, modified OpenLTE and YateBTS 5.5.0 for LTE and GSM network implementation respectively. The experimental setup for the GSM network is shown in Figure 7 (Appendix).

Ethical Considerations. During our base station experiments, we used a Faraday cage whenever possible. If a Faraday cage was not available, we ensured that we did not interfere and transmit on any frequencies used by local base stations. To meet this goal, we used non-overlapping frequency in the region, set the transmission power low enough to prevent external devices from connecting, and confirmed that other nearby devices are not affected by our experiment.

LTE RRC. For the verification of crashes in LTE, we modified the OpenLTE source code to transmit the FIRMWIRE generated payload. Specifically, the function encoding *RRCConfigurationReconfiguration* message was modified according to the test case LTE RRC #1 to #3, and the state machine is modified to transmit this signaling message directly after when a UE attaches to the modified base station, without authentication. We skipped the test case LTE RRC #4 because OpenLTE does not support MCCH signaling, which is used only by eMBMS (Evolved Multimedia Broadcast Multicast Services). According to the GSA, only 5 cellular network operators launched a commercial eMBMS service in 2019 [24], making this issue less relevant in the real world at present. Although alternatives such as srsRAN and OpenAirInterface do support eMBMS, we leave such testing to future work.

GSM SM, CC. For the verification of crashes in GSM, we modified the YateBTS source code to transmit the FIRMWIRE generated payload. Specifically, the function encoding PCO structure was modified for GSM SM, and the function encoding *Call Setup* was modified for GSM CC.

Device Preparation. We modified the cellular network software as described above for the over-the-air testing. Our test devices are Galaxy S7 EDGE, S8, S9, S10, and S10e. We allowed the device to connect to our base station (either LTE or GSM), and checked whether each of our crashes can be reproduced over-the-air:

LTE RRC: We wait for the UE to attach to the base station. The base station sends our payload, and the baseband crashes.

GSM SM: We wait for the UE to acquire a GPRS PDP context by activating a data connection. The base station sends our payload, and the baseband crashes.

GSM CC: We initiate a call from another UE. Corruption occurs when the target UE receives and answers the call.

For all test cases, the crash was visually confirmed when the cellular signal bars disappeared and via ADB radio logs with the `CP crash` indicator.

Over-the-air testing of a variety of crashes for multiple firmware versions is both tedious and time-consuming work, as 1) baseband firmware is often linked to the specific version of Android system firmware, requiring the time-consuming whole firmware flashing and 2) upgrading and downgrading to any arbitrary version is not always possible due to anti-rollback protections. Hence, we performed over-the-air testing on a small number of devices and firmware images. Our intuition is that over-the-air reproduction of crashing inputs found during emulation can serve as a proxy for FIRMWIRE's accuracy.

Results. We were able to reproduce LTE RRC #1 and #3 over-the-air against a Galaxy S10e (G970FXXU3BSKL). This is in line with our emulation results, as this device model is not affected by the crashing input for LTE RRC #2. Similarly, we could observe LTE RRC #1 for a Galaxy S9 (G960FXXS7CSJ3), which is not vulnerable to LTE RRC #2 and #3 according to FIRMWIRE. Corresponding to our threat model, the phone does not require to be fully attached to the base station, as the signaling message can be sent just after the attach request. We also note that this message is visually indistinguishable from network traffic sent by other base stations in the Android user interface, and can cause a Denial of Service (DoS) for other phones in the rogue base station's operation range.

We further reproduced selected CC and SM crashes over-the-air. While we did not have access to S8, S9, or S10e phones with old firmware vulnerable to all three findings, we could replicate CC #1 and SM #1 against a Galaxy S7 EDGE (G935FXXU1DPCT). We further replicated CC #2 against an additional S10 (G973FXXS5CTD1). In all of these test cases, the phone needs to be attached to the attacker-controlled rogue base station, which is a realistic attack scenario for these vulnerabilities, as 2G lacks mutual authentication.

All of these experiment results confirm our findings during emulation, and, hence, we believe that FIRMWIRE's results are a good indicator for the behavior of real phones.

Discussion. These end-to-end examples demonstrate the power and usability of FIRMWIRE to increase the productivity of baseband research. If an analyst were to try and find the flaws mentioned using over-the-air fuzzing alone, they would have to continuously initiate calls and GPRS activations to the device, with test rates close to 1 to 2 tests per minute. Additionally, due to the statefulness of the cellular protocols, they would have to not only write code to achieve this for every other message type but also physically reset the device if the internal state had been corrupted, preventing further testing. Even if a crash occurred, with little insight into the root cause of the crash beyond the message sent, debugging the root cause would be nearly impossible. Using FIRMWIRE's scalable protocol testing (**Q1** & **Q2**) and instrumentation not offered by production hardware, we are able to get insight into memory corruptions and the overall operation and resilience of real-world baseband implementations.

VI. DISCUSSION

Supporting Qualcomm Basebands. The largest vendor for baseband platforms by revenue, Qualcomm, poses an additional challenge to FIRMWIRE. Unlike other baseband implementations, Qualcomm leverages a fully-custom ISA known as Hexagon, specifically built for their Digital Signal Processors (DSPs). Unfortunately, tooling for this ISA is sparse, and especially full-system emulators are lacking. To date, only rudimentary user-mode emulation frameworks are publicly available for Hexagon [53]. As a result, the most difficult hurdle in supporting Qualcomm BPs would be implementing a full-system emulator for an additional ISA, which is outside the scope of this work. However, given such an emulator, a vendor plugin could be created to add support to FIRMWIRE for Hexagon's hardware and RTOS (QuRT).

Supporting 5G Basebands. During our research, we also performed an initial assessment of Samsung's 5G modem (the S5123 chipset). The most notable change is in the hardware platform: instead of using a Cortex-R processor, the new chipset uses the Cortex-A series. Early investigation into the binary shows that the ShannonOS is nearly the same structure, with the major changes centered around the memory layout. Firmware for the MT6853 and MT6873 MediaTek chipsets present a similar situation, with new nanoMIPS-based BPs which require vendor plugin updates. In both cases, the core RTOS primitives appear unchanged, which indicates that we can easily reuse most of the existing FIRMWIRE vendor plugins. We plan to apply FIRMWIRE to these additional hardware platforms in future work in order to explore the newly created 5G NR implementations.

Symbolic Execution. One of the benefits of our emulation-based approach is that it allows us to run security analysis tooling that would have been otherwise inaccessible. One example is plug-and-play symbolic analysis of the firmware using concrete state extracted from the emulator [23], [49]. We implemented a proof of concept on top of the angr framework [62], in which we snapshot the emulator state and transfer it to the symbolic execution engine. Then, an analyst can annotate which parts of the memory should be made symbolic, e.g., function inputs or memory buffers, before beginning the symbolic exploration.

Thanks to the rich concrete state provided by the emulator, typical problems such as state explosion and over-approximation impact the symbolic exploration in our proof of concept less severely than approaches starting from a fully symbolic state, such as BaseSpec [38]. However, we note that for such an analysis, the emulated baseband and included state machines need to be driven towards the kick-off point for symbolic exploration, which we consider to be an independent challenge outside the scope of this work.

Exploring other Interfaces. To enable a fully virtual UE, USIM peripheral support would need to be prototyped into FIRMWIRE. This would yield more accurate processing of messages, especially those which require a SIM card such as SMS or USSD. Basebands also have many other protocols, including GPS/GNSS, audio codecs, AT commands, diagnostic, and remote IPC. Exploring how basebands decode GPS over-the-air could uncover new vulnerabilities beyond the cellular standards. Audio codecs are complex on their own and could be investigated further to make sure they operate with malformed voice packets controlled by callers. AT command interfaces are a large attack surface on some devices [65], [37]. Exploring the many AT command handlers could yield a better understanding of this attack surface. Diagnostic modes [30], similar to AT commands, are highly privileged and less explored, yet yield significant power for baseband introspection, without requiring advanced dynamic analysis. Finally, remote IPC from the application processor and system processes running on the AP, is a large attack surface that needs more auditing. For example, how the baseband can affect the AP's kernel and its structures is of great interest in the event that the baseband is compromised by remote attack.

More than Introspection. FIRMWIRE's application to baseband introspection and security testing is clear. But, many uses outside of the security community exist. This includes the ability to have a fully virtualized UE for use in operating system emulators, such as those for Android and iOS, or in continuous integration environments for Radio Access Networks (RANs). For example, in typical mobile emulators, the baseband processor does not exist and is instead stubbed out. This means the environment is not able to test flows involving calls or text messages. Instead, these have to be tested on real devices, which prevents easy automation. Additionally, open source RANs like srsRAN, OpenAirInterface, YateBTS, and more could integrate FIRMWIRE into their CI environment. This could be achieved by mocking out the physical layer and instead tunneling layer-2 packets to and from a RAN. srsRAN offers this functionality out-of-the-box via its ZeroMQ interface.

VII. Related Work

Cellular Protocol Security. BaseSpec [38] by Kim et al. performs static and symbolic analysis of baseband firmware to extract protocol models. It then compares these to the cellular standards to discover inconsistencies. While systematic, this approach suffers from false positives due to unconstrained symbolic execution, only considers NAS protocols, and does not take into consideration accurate baseband state. FirmWire is a dynamic analysis platform for basebands that has few false positives, supports a multitude of protocols (not just NAS), and supports stateful testing. LTEFuzz [40] tests LTE NAS and Radio Resource Control (RRC) message handling on core and user equipment over-the-air. Their approach inverts protocol decision trees in order to classify invalid test cases and mutate a corpus of LTE control packets collected using SCAT [30]. DoLTEst [55] offers a more systematic approach to downlink testing, but like LTEFuzz, also requires physical devices with log monitoring enabled. Our work takes a different approach by emulating the device's baseband processor to explore the fine-grained impact of message parsing and memory corruption bugs. Using FirmWire, we are able to pinpoint the precise reason for a crash, which would normally be externally considered as a "denial of service", and use our debugging capabilities to further understand the security impact. Additionally, LTEFuzz effectively uses black-box fuzzing, while FirmWire provides coverage-guided fuzzing, increasing code coverage while avoiding manual effort. Most notably, neither LTEFuzz nor BaseSpec are designed as baseband analysis platforms and neither released reproducible artifacts (full source code and data set) to the community, limiting their usefulness.

T-Fuzz [34] uses existing cellular test suites based on TTCN-3 as a basis for negative testing (fuzzing) of cellular protocols. It is unclear what devices are tested and there is no mention of this work being used for basebands. Besides fuzzing, many papers focus primarily on cellular protocols and standards. LTEInspector [31] discovered multiple flaws within LTE by focusing on protocol state machines and their security properties. 5GReasoner [32] took a similar approach for 5G using model checking, leading to multiple weaknesses and vulnerabilities. Atomic [10] analyzes the wording of LTE specification documents, finding vulnerabilities in LTE itself as well as some implementations. A large-scale analysis of VoLTE demonstrated many attacks on network operators [39]. The ephemeral GUTI identifier, used in LTE to page specific mobile devices while maintaining over-the-air privacy, was shown to have bad random properties, allowing tracking of subscribers [29]. The aLTEr attack on the LTE MAC layer demonstrated cryptographic flaws with the Authentication and Key Agreement (AKA) and ciphertext attacks, which allowed for IP traffic and device fingerprinting [58], downgrading, and more. Shaik et al. [61] and Chlosta et al. [11] also demonstrate LTE downgrade attacks, UE fingerprinting, battery draining attacks, and unprotected RRC messages. SigOver [69] targets the LTE physical channel to craft malicious broadcast messages, signaling storms, denial of service, and downgrades. SigUnder [44] attacks the 5G NR physical channel to enable similar capabilities to the SigOver family of attacks.

Baseband Firmware Security. Early work targeting the cellular baseband in 2009 by Mulliner et al. [51] demonstrated generating invalid SMS messages through fuzzing on a local device to reveal parsing and protocol flaws. This work was extended years later to demonstrate that remote SMS attacks against mobile devices are possible [50]. Following work in 2012 demonstrated that even more damaging remote attacks are possible over-the-air as a result of targeted baseband memory corruption [67]. This highlighted the relative insecurity of baseband processors when compared to application processors, due to missing mitigations against memory corruption, such as Address Space Layout Randomization (ASLR) [60]. This work was limited to two phones and relied primarily on manual static analysis. Recent work on baseband exploitation [54], [6] confirms that practical remote attacks are still possible and inexpensive to mount. Testing baseband firmware using fuzzing was shown in [66], [28] which focuses on GSM, but only targets SMS and SMS cell broadcast. Previous approaches to scaling up cellular testing include developing a wireless testbed and incorporating feedback from multiple UEs to detect faults [17], [27]. Most previous work performs over-the-air testing, which is difficult to fully scale and relies on manufacturer logs from UEs, which may not be detailed enough or available at all, to triage faults. Notable, very recent, exceptions exist. One example, alas not targeting cellular basebands, is Frankenstein by Ruge et al., [57] proposing to rewrite parts of a Bluetooth baseband. With their methodology, they are able to run, and fuzz, the Bluetooth baseband on user-mode QEMU after extracting a snapshot from a physical device. Targeting cellular basebands, BaseSAFE fuzzes cellular basebands by rehosting single functions to Unicorn, a CPU emulator, and executing them directly [45]. As outlined in Section V-C, BaseSAFE requires the manual creation of an initial state and can benefit from snapshots created with FirmWire.

Firmware Rehosting. Recently, rehosting for security testing gained a lot of traction [18], [68]. Hardware-in-the-loop approaches [36], [70], [42], [13], [25], [63] emulate target firmware while forwarding hardware interaction to a physical device. This does not only pose inherent challenges to scalability but requires also the presence of advanced debug interfaces capable of memory introspection and modification. Unfortunately, due to the locked-down nature of cellular processors, such interfaces are rarely available, rendering hardware-in-the-loop approaches insufficient for this line of research.

To avoid direct hardware interactions and the resulting challenges, various hardware-less rehosting solutions make use of known and well-defined abstractions such as the Linux kernel [9], [14], [71], [41], or hardware abstraction layers [12], [43]. Furthermore, to rehost targets where hardware accesses cannot be eliminated, recent approaches deploy simple models [26], [19], [72], [35] to create peripheral mock-ups and enable dynamic firmware analysis without a physical device, and a recent work extends these approaches to add support for simple DMA transactions [47]. Unfortunately, due to its obscurity and complexity, baseband firmware poses a combination of challenges which previous approaches handle strictly separately: It is a highly complex, proprietary piece of software that is not based on Linux and for which no public SDKs exist (as would be needed for library matching schemes such as HALucinator [12]). At the same time, its input channels rely on memory sharing with the AP and DSPs, as well as complex DMA interactions (e.g., via ring buffers). Both types of input channels are not handled by the current state of the

art for automated hardware modeling. As such, no previous approach cleanly applies to the complex area of baseband firmware analysis.

VIII. Conclusion

In this paper, we presented FirmWire, a scalable, full-system emulation platform for baseband firmware analysis, supporting firmware from two major device vendors spanning 6 different chipsets. Using FirmWire, we discovered 4 previously unknown pre-authentication vulnerabilities in LTE and GSM protocols via coverage-guided fuzzing. By replaying the crashing inputs against 213 emulated firmware images, we gained detailed insight into patching cycles and vulnerability lifetimes. Finally, we demonstrated the accuracy of FirmWire's emulation by replicating the crashes over-the-air on multiple devices.

Acknowledgments

We want to thank the anonymous reviewers for their insightful comments and feedback. Additionally, we want to express our gratitude to Samsung for working with us on the identified issues and providing patches to the end-users in a timely manner.

This work was partially supported by the US National Science Foundation grant CNS-1815883, the Office of Naval Research grant ONR-OTA N00014-20-1-2205, the Air Force Office of Scientific Research award FA9550-14-1-0351, the Semiconductor Research Corporation, the Netherlands Organisation for Scientific Research through grants NWO "TROPICS" (628.001.030) and NWA-ORC "InterSect", and the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] 3GPP, "Call Barring (CB) supplementary service; Stage 3," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 24.088. [Online]. Available: http://www.3gpp.org/DynaReport/24088.htm

[2] ——, "Call Forwarding (CF) supplementary services; Stage 3," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 24.082. [Online]. Available: http://www.3gpp.org/DynaReport/24082.htm

[3] ——, "Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 36.331. [Online]. Available: http://www.3gpp.org/DynaReport/36331.htm

[4] ——, "Mobile radio interface Layer 3 specification; Core network protocols; Stage 3," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 24.008. [Online]. Available: http://www.3gpp.org/DynaReport/24008.htm

[5] ——, "Mobile radio interface layer 3 supplementary services specification; Formats and coding," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 24.080. [Online]. Available: http://www.3gpp.org/DynaReport/24080.htm

[6] Amat Cama, "A Walk with Shannon: Walkthrough of a Pwn2Own Baseband Exploit," in *Infiltrate*, Apr. 2018. [Online]. Available: https://downloads.immunityinc.com/infiltrate2018-slidepacks/amat-cama-a-walk-with-shannon/presentation.pdf

[7] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "IJON: Exploring deep state spaces via fuzzing," in *IEEE Symposium on Security and Privacy*, 2020.

[8] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX Annual Technical Conference*, 2005.

[9] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware." in *Symposium on Network and Distributed System Security (NDSS)*, 2016.

[10] Y. Chen, Y. Yao, X. Wang, D. Xu, C. Yue, X. Liu, K. Chen, H. Tang, and B. Liu, "Bookworm Game: Automatic Discovery of LTE Vulnerabilities Through Documentation Analysis," in *IEEE Symposium on Security and Privacy*, 2021.

[11] M. Chlosta, D. Rupprecht, T. Holz, and C. Pöpper, "LTE security disabled: misconfiguration in commercial networks," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[12] A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *USENIX Security Symposium*, 2020.

[13] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-Wide Security Testing of Real-World Embedded Systems Software," in *USENIX Security Symposium*, 2018.

[14] A. Costin, A. Zarras, and A. Francillon, "Automated dynamic firmware analysis at scale: a case study on embedded web interfaces," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2016.

[15] Counterpoint Research, "Smartphone application processor (AP) / system-on-chip (SoC) vendor shipment share worldwide in 2020 and 2021," 2021.

[16] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with PANDA," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, 2015.

[17] K. Fang and G. Yan, "Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning," in *European Symposium on Research in Computer Security (ESORICS)*. Springer International Publishing, 2018.

[18] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *et al.*, "SoK: Enabling Security Analyses of Embedded Systems via Rehosting," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[19] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling," in *USENIX Security Symposium*, 2020.

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[21] N. Golde and D. Komaromy, "Breaking Band: Reverse Engineering and Exploiting the Shannon Baseband," in *RECON*, Jun. 2016. [Online]. Available: https://comsecuris.com/slides/recon2016-breaking_band.pdf

[22] M. Grassi and Kira, "Exploring the MediaTek baseband," in *OffensiveCon*, Feb. 2020.

[23] F. Gritti, L. Fontana, E. Gustafson, F. Pagani, A. Continella, C. Kruegel, and G. Vigna, "SYMBION: Interleaving Symbolic with Concrete Execution," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, 2020.

[24] GSA, "LTE Broadcast (eMBMS) Market Update," Jul. 2019.

[25] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the Analysis of Embedded Firmware through Automated Re-hosting," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[26] L. Harrison, H. Vijaykumar, R. Padhye, K. Sen, and M. Grace, "PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation," in *USENIX Security Symposium*, 2020.

[27] G. Hernandez and K. R. B. Butler, "Basebads: Automated security analysis of baseband firmware: poster," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. Association for Computing Machinery, 2019.

[28] B. Hond, "Fuzzing the GSM protocol," Master's thesis, Radboud University Nijmegen, Netherlands, Jul. 2011.

[29] B. Hong, S. Bae, and Y. Kim, "GUTI Reallocation Demystified: Cellular Location Tracking with Changing Temporary Identifier." in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[30] B. Hong, S. Park, H. Kim, D. Kim, H. Hong, H. Choi, J.-P. Seifert, S.-J. Lee, and Y. Kim, "Peeking Over the Cellular Walled Gardens - A Method for Closed Network Diagnosis -," *IEEE Transactions on Mobile Computing*, Oct. 2018.

[31] S. R. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE," in *Symposium on Network and Distributed System Security (NDSS)*, 2018.

[32] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino, "5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[33] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on ARM disassembly tools," in *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020.

[34] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano, "T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols," in *IEEE Transactions on Control System Technology (CST)*, 2014.

[35] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, "Jetset: Targeted Firmware Rehosting for Embedded Systems," in *USENIX Security Symposium*, 2021.

[36] M. Kammerstetter, C. Platzer, and W. Kastner, "PROSPECT: Peripheral Proxying Supported Embedded Code Testing," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.

[37] I. Karim, F. Cicala, S. R. Hussain, O. Chowdhury, and E. Bertino, "Opening Pandora's Box through ATFuzzer: Dynamic Analysis of AT Interface for Android Smartphones," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[38] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "BaseSpec: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[39] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim, "Breaking and Fixing VoLTE: Exploiting Hidden Data Channels and Mis-implementations," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[40] H. Kim, J. Lee, E. Lee, and Y. Kim, "Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane," in *IEEE Symposium on Security and Privacy*, 2019.

[41] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis," in *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[42] K. Koscher, T. Kohno, and D. Molnar, "SURROGATES: Enabling near-real-time dynamic analyses of embedded systems," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[43] W. Li, L. Guan, J. Lin, J. Shi, and F. Li, "From Library Portability to Para-rehosting: Natively Executing Microcontroller Software on Commodity Hardware." in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[44] N. Ludant and G. Noubir, "SigUnder: a Stealthy 5G Low Power Attack and Defenses," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2021.

[45] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband SAnitized Fuzzing through Emulation," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2020.

[46] Marco Grassi, Muqing Liu, and Tianyi Xie, "Exploitation Of A Modern Smartphone Baseband," in *Black Hat USA*, 2018.

[47] A. Mera, B. Feng, L. Lu, and E. Kirda, "DICE: Automatic emulation of dma input channels for dynamic firmware analysis," in *IEEE Symposium on Security and Privacy*, 2021.

[48] G. Miru, "Path of Least Resistance: Cellular Baseband to Application Processor Escalation on Mediatek Devices," Jul. 2017, Comsecuris Blog.

[49] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar²: A Multi-target Orchestration Platform," in *Workshop on Binary Analysis Research (BAR)*, 2018.

[50] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale," in *USENIX Security Symposium*, 2011.

[51] C. Mulliner and C. Miller, "Fuzzing the Phone in your Phone," in *Black Hat USA*, 2009.

[52] NCC Group, "TriforceAFL," 2017. [Online]. Available: https://github.com/nccgroup/TriforceAFL

[53] Niccolò Izzo and Taylor Simpson, "QEMU-Hexagon: Automatic Translation of the ISA Manual Pseudcode to Tiny Code Instructions of a VLIW Architecture," KVM Forum, Nov. 2019.

[54] Nico Golde, "There's Life in the Old Dog Yet: Tearing New Holes into Intel/iPhone Cellular Modems," Apr. 2018, comsecuris Blog.

[55] C. Park, S. Bae, B. Oh, J. Lee, E. Lee, I. Yun, and Y. Kim, "DoLTEst: In-depth Downlink Negative Testing Framework for LTE Devices," in *USENIX Security Symposium*, 2022.

[56] N. A. Quynh and D. H. Vu, "Unicorn: Next Generation CPU Emulator Framework," *BlackHat USA*, 2015.

[57] J. Ruge, J. Classen, F. Gringoli, and M. Hollick, "Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets," in *USENIX Security Symposium*, 2020.

[58] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Breaking LTE on Layer Two," in *IEEE Symposium on Security and Privacy*, 2019.

[59] SamMobile, "Download firmware updates for your Samsung mobile phone and tablet," 2021. [Online]. Available: https://sammobile.com/firmwares/

[60] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-space Randomization," in *ACM Conference on Computer and Communications Security (CCS)*, 2004.

[61] A. Shaik, R. Borgaonkar, S. Park, and J.-P. Seifert, "New vulnerabilities in 4G and 5G cellular access network protocols: exposing device capabilities," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2019.

[62] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "SoK:(state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[63] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems," in *USENIX Security Symposium*, 2018.

[64] T. B. Team, "Exploring Qualcomm Baseband via ModKit," in *CanSecWest*, 2018.

[65] D. Tian, G. Hernandez, J. Choi, V. Frost, C. Ruales, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace, and K. Butler, "ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem," in *USENIX Security Symposium*, 2018.

[66] F. van den Broek, B. Hond, and A. Cedillo Torres, "Security Testing of GSM Implementations," in *Engineering Secure Software and Systems*, J. Jürjens, F. Piessens, and N. Bielova, Eds. Springer International Publishing, 2014.

[67] R.-P. Weinmann, "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks." in *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[68] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in Firmware Re-Hosting, Emulation, and Analysis," *ACM Computing Surveys (CSUR)*, 2021.

[69] H. Yang, S. Bae, M. Son, H. Kim, S. M. Kim, and Y. Kim, "Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE," in *USENIX Security Symposium*, 2019.

[70] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[71] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *USENIX Security Symposium*, 2019.

[72] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Automatic Firmware Emulation through Invalidity-guided Knowledge Inference," in *USENIX Security Symposium*, 2021.

Appendix

PatternDB Entry Format

```
Pattern := {
  name := string
  pattern := [ PatternSyntax... ]
  lookup := PatternFn?
  post_lookup := PatternFn?
  required := bool?
  for := [ string... ]?
  within := [ AddressSet... ]?
  offset := integer?
  offset_end := integer?
  align := integer?
}

PatternSyntax :=
    r"([a-fA-F0-9]{2}|([?][?+*]))+"
PatternFn := code
AddressSet := SymbolName | AddressRange
SymbolName := string
AddressRange := [integer, integer]
```

The syntax above uses `:=` for type definitions, `[]` for arrays, `...` for one or more of a type, `|` for either or, and `?` for optional elements. `Pattern` is made up of two required fields: a name and one or more `PatternSyntax` elements. `PatternSyntax` is a plain-text string that abstracts away from regular expressions. This string is made up of hex bytes, masked bytes `??`, variable-length byte sequences using `?*` for zero or more and `?+` for one or more. For more complex patterns (e.g. ones requiring Turing complete checks), the `lookup` field can point to a code sequence. Patterns can also have a post-processing function (`post_lookup`) after a match has been found. FirmWire provides built-in post-processing helpers to perform operations such as searching for pointer references and dereferencing pointers. In addition, there are other optional fields like `required`, which make pattern failure halt execution, `for` which constrains a pattern to specific devices, `within` which only searches for patterns within a symbol region or address range (the default is the entire address range), `offset` and `offset_end` which adjust the final address, and finally `align` which constrains found addresses to a byte alignment.

To develop a pattern, an analyst would use a static analysis tool to identify an area of interest within one or more firmware images. They would then discover nearby byte sequences and using hex patterns, lookup functions, and constraints test the pattern on the target set of binaries, ensuring the same relative location is matched. Effectively, PatternDB performs light-weight static analysis and *does not* require an external static analysis tool or disassembly to operate.

Implementation Details

We provide the full list of used patterns for the Samsung and MediaTek vendor plugins, as well as implemented peripheral models in Table VII, Table VIII, and Table IX below.

Selecting Proxies for per-task Coverage

Retrieving an accurate number for the achieved coverage for individual tasks in the baseband firmware images is a non-trivial problem. Firstly, these firmware images are only distributed in binary form, usually spanning 20-50MB in size. As such, to retrieve the correct number of basic blocks embedded in this binary, perfect disassembly of the binary would be required, which is an undecidable problem for architectures interleaving code and data. Additionally, even if source code was available, establishing boundaries between the tasks is difficult due to shared code and common API calls. Lastly, even when considering non-perfect disassembly, a recent study demonstrates the limitations of state-of-the-art disassembly tools for ARM binaries [33], which largely reflects our experiences for analyzing baseband modems.

To this end, we deploy two proxies for a best-effort mapping between fuzzed tasks and associated basic blocks. For MediaTek images, we leverage debugging symbols shipped together with the baseband firmware. These symbols are stored in a proprietary format and contain start addresses and names of functions. We filter these functions by the prefix likely associated with the fuzz task (i.e., *errc*) and consider all basic blocks of the corresponding disassembled functions as relevant code.

For Samsung images, we use the debugging trace entries embedded in the modem binaries. These trace entries are located in a specific region of the loaded firmware, follow a well-specified format, and hold a pointer to a string for the associated source code file name. Therefore, we could easily discover the trace entries and associate them to the specific tasks, based on the file name. Then, for each trace entry associated with a task, we automatically check for references to the trace entry in the disassembled binaries. If a reference is found, we automatically attempt to identify the function boundaries for the code using this reference and consider all basic blocks in resulting functions as relevant.

Over-the-air testing setup

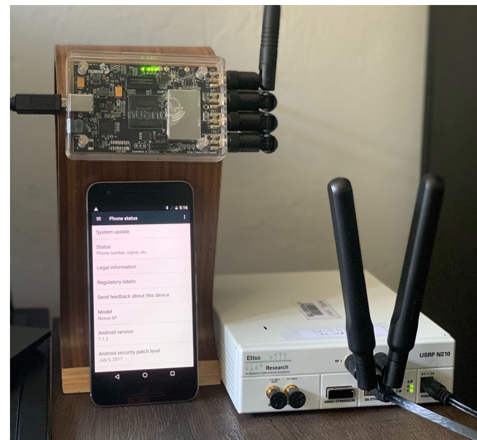


Fig. 7: Hardware setup for YateBTS GSM network. A similar setup had been used for the OpenLTE LTE network.

Name	Pattern/Lookup	PostLookup	Req?	For	Within	Offset	OffsetEnd	Align
boot_mpu_table	`00000000 00000000 1c000000???????? ???????? ???????? ???????? ???????? ???????? 01000000 01000000 00000004 20`	-	T	*	*	0x0	0x0	0x0
boot_setup_memory	`PAT1: 00008004 200c0000` `PAT2: 00000004 ????0100`	`parse_memory_table`	T	*	*	-0x14	0x0	0x4
boot_key_check	`?? 49 00 22 ?? 48 ?? a3 ?? ?? ?? ?? 80 21 68 46 ?? ?? ?? ?? 10 22 20 a9 68 46 ?? ?? ?? ??`	-	T	S5000AP	*	0x0	0x0	0x0
OS_fatal_error	`70 b5 05 46 ???????? ?? 48 ?? 24`	-	F	*	*	0x0	0x0	0x0
pal_MemAlloc	`2d e9 f0 4f 0d 00 83 b0 99 46 92 46 80 46`	`fixup_bios_symbol`	F	*	*	0x0	0x0	0x0
pal_MsgSendTo	`PAT1: 70 b5 ?+ 04 46 15 46 0e 46 ?? ?? 01 df ?* 88 60 08 46 ?+ ?? 48 ???? ???? 20 46 98 47` `PAT2: ???????? b0f5fa7f 0446 ??46`	-	F	*	*	0x0	0x0	0x0
pal_Sleep	`30 b5 ?+ 98 ?+ ??98 ??22 ??23 11 46 ?? 94`	-	F	*	*	0x0	0x0	0x0
log_printf	`0fb4 2de9f047 ???? ??98 d0e90060 c0f34815`	-	T	*	*	0x0	0x0	0x0
log_printf2	`0fb4 2de9f04f ???? ??0a 8fb01898 4068`	-	F	*	*	0x0	0x0	0x0
pal_SmSetEvent	`PAT1: 10b5 ???? ???????? 04 b2` `PAT2: 10b5 0068 0028 ???? ???????? 04 b2`	-	F	*	*	0x0	0x0	0x0
SYM_EVENT_GROUP_LIST	`70 40 2d e9 00 40 a0 e1 ?? ?? 00 eb 00 50 a0 e1 20 00 9f e5 04 10 a0 e1 ?? 05 00 eb ?? 00 94 e5 00 00 50 e3 30 ff 2f 11 05 00 a0 e1 ?? ?? 00 eb 00 00 a0 e3 70 80 bd e8`	`dereference`	F	*	*	0x0	0x0	0x0
SYM_TASK_LIST	`find_task_table`	-	F	*	*	0x0	0x0	0x0
SYM_SCHEDULABLE_TASK_LIST	`find_schedulable_task_table`	-	F	*	*	0x0	0x0	0x0
SYM_CUR_TASK_ID	`find_current_task_ptr`	-	F	*	*	0x0	0x0	0x0
SYM_FN_EXCEPTION_SWITCH	`find_exception_switch`	-	F	*	*	0x0	0x0	0x0
SYM_QUEUE_LIST	`find_queue_table`	-	F	*	*	0x0	0x0	0x0
SXXXAP_DVFS	`PAT1: ??f8???? 00f01f01 ??48 d0 f8 ???? c0 f3 ???? ???????? ???? 00 ?? ?* ??f1???? ??82 ??eb??11 0988` `PAT2: ???? 00 ?? ?* ??f1???? ??82 ??eb??11 0988`	-	T	S335AP, S355AP, S360AP	*	0x0	0x0	0x2
SYM_LTERRC_INT_MOB_CMD_HO_FROM_IRAT_MSG_ID	`find_lterrc_int_mob_cmd_ho_from_irat_msgid`	-	F	*	*	0x0	0x0	0x0

TABLE VII: PatternDB for Shannon baseband images

Name	Pattern/Lookup	PostLookup	Req?	For	Within	Offset	OffsetEnd	Align
INC_Initialize_corewait	`409a 609b 62ea`	-	F	*	`INC_Initialize`	0x0	-0x2	0x0
sync1_addr_code	`609a 809c 82eb`	-	F	*	`INC_Initialize`	-0x2	0x0	0x0
nvram_ltable_init_code	`a0e8`	-	F	*	`nvram_ltable_construct`	0x0	0x0	0x0
corewait_addr_code	`11ea 6eea 2367`	-	F	*	`stack_init_tasks`	0x0	0x0	0x0
L1D_ CustomDynamicGetParam_ assert	`00 f1 00 5c`	-	F	*	`L1D_ CustomDynamicGetParam`	-0x2	0x0	0x0
errc_evth_inevt_handler_ assert	`65 ea`	-	F	*	`errc_evth_inevt_handler`	0x0	0x0	0x0
errc_evth_inevt_handler_ end	`???? a0e8`	-	F	*	`errc_evth_inevt_handler`	0x0	0x0	0x0
ptr_logical_data_item_ table	`a0 e8`	-	F	*	`nvram_util_get_data_item`	0x0	0x0	0x0
ptr_sys_comp_config_tbl	`a0 e8`	-	F	*	`SST_CheckHealthinessQCB`	0x0	0x0	0x0

TABLE VIII: PatternDB for MediaTek baseband images

Peripheral	Description	Used by	SLoC
CyclicBitPeripheral	Dummy peripheral returning different bit set on every access	S*	14
LoggingPeripheral	Generic peripheral logging read and write accesses	S*	15
PassThrough	Peripheral behaving like ordinary memory	M*	50
GLink	Custom peripheral injected by FIRMWIRE for interaction with an emulated system	-	501
DSP	Digital Signal Processor for cellular messages. Firmware expects correct sync words	S*	22
PMIC	Power Management Integrated Circuit, requires platform specific contents	S355AP, S360AP, S5000AP	37
S355APClk	Core system clock for S355AP chipsets	S335, S355	92
S355DSPBuffer	DSP Buffer for S355AP	S355AP	16
S360APClk	Core system clock for S360AP chipsets	S360	55
S3xxAPBoot	Peripheral returning specific values for some hardware platforms during boot	S335AP, S360AP	19
S5000APClk	Core system clock for S5000AP chipsets	S5000AP	34
ShannonAbox	Mock-Up for audio related peripheral	S*	23
ShannonSOC	Peripheral containing core information over the SoC	S*	40
ShannonTCU	Timer Control Unit for various timers	S*	24
ShannonTimer	Custom timer. Due to frequent accesses implement directly in QEMU.	S*	177
SHM	Shared Memory to communicate with the AP via circular FIFO buffers	S*	157
SIPC	Peripheral for IPC between AP and CP	S*	38
UARTPeripheral	Used during boot for I/O	S*	18
Unknown2	Peripheral who is expected to return a specific value	S*	15
AES	Hardware peripheral for AES	M*	45
CDMM	Dummy Peripheral for Common Devyce Memory Map	M*	3
GCR	Global Configuration Registers	M*	25
MCUSync	Dummy peripheral	M*	3
MDC	Modem Configuration Peripheral	M*	59
MDPERISYS_MISC	Dummy peripheral for synchronization with AP	M*	4
MODEML1_TOPSM	Dummy peripheral return 0xffffffff at offset 0xD4	M*	9
OSTimer	Timer peripheral dedicated to the OS	M*	40
PCCIF	Peripheral for communicating with the AP via Shared Memory	M*	486
PMIC	Power Management Integrated Circuit	M*	51
TDMABase	Timer peripheral used for TDMA	M*	10
TOPSM	Dummy peripheral return 0xffffffff at offset 0x590	M*	8

TABLE IX: List of peripherals created for FIRMWIRE and platforms using them.
S*: Peripheral is used for all Shannon images. **M*:** Peripheral is used for all MediaTek images.