

BaseSAFE: Baseband SANitized Fuzzing through Emulation

Dominik Maier
dmaier@sect.tu-berlin.de
TU Berlin

Lukas Seidel
seidel.1@campus.tu-berlin.de
TU Berlin

Shinjo Park
pshinjo@sect.tu-berlin.de
TU Berlin

ABSTRACT

Rogue base stations are an effective attack vector. Cellular basebands represent a critical part of the smartphone's security: they parse large amounts of data even before authentication. They can, therefore, grant an attacker a very stealthy way to gather information about calls placed and even to escalate to the main operating system, over-the-air. In this paper, we discuss a novel cellular fuzzing framework that aims to help security researchers find critical bugs in cellular basebands and similar embedded systems. BaseSAFE allows partial rehosting of cellular basebands for fast instrumented fuzzing off-device, even for closed-source firmware blobs. BaseSAFE's sanitizing drop-in allocator, enables spotting heap-based buffer-overflows quickly. Using our proof-of-concept harness, we fuzzed various parsers of the Nucleus RTOS-based MediaTek cellular baseband that are accessible from rogue base stations. The emulator instrumentation is highly optimized, reaching hundreds of executions per second on each core for our complex test case, around 15k test-cases per second in total. Furthermore, we discuss attack vectors for baseband modems. To the best of our knowledge, this is the first use of emulation-based fuzzing for security testing of commercial cellular basebands. Most of the tooling and approaches of BaseSAFE are also applicable for other low-level kernels and firmware. Using BaseSAFE, we were able to find memory corruptions including heap out-of-bounds writes using our proof-of-concept fuzzing harness in the MediaTek cellular baseband. BaseSAFE, the harness, and a large collection of LTE signaling message test cases will be released open-source upon publication of this paper.

CCS CONCEPTS

• **Security and privacy** → *Software reverse engineering*; **Embedded systems security**.

KEYWORDS

fuzzing, cellular, security, rehosting

ACM Reference Format:

Dominik Maier, Lukas Seidel, and Shinjo Park. 2020. BaseSAFE: Baseband SANitized Fuzzing through Emulation. In *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, July 8–10, 2020, Linz (Virtual Event), Austria. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3395351.3399360>

WiSec '20, July 8–10, 2020, Linz (Virtual Event), Austria

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20)*, July 8–10, 2020, Linz (Virtual Event), Austria, <https://doi.org/10.1145/3395351.3399360>.

1 INTRODUCTION

Attacks on mobile basebands are possible from adjacent base stations, which are built to work over fairly large distances. The targets move constantly, carrying their phone with them, so a rogue base station can potentially attack a large number of phones every day. Even though operating systems running on the phone's basebands are large attack vectors, parsing many different types of signaling messages, even prior to authentication, the systems are obscure and little research exists. Each larger chipset vendor for smartphones develops and ships their own stack. Any sort of automated analysis, if it exists at all, is kept locked away behind the vendor's doors, together with documentation and specifications of their systems. This gave us the reason to take a closer look at one of them. Since the systems are written in C/C++, known to be haunted by memory corruptions, we set out to build a usable open-source fuzzing environment, BaseSAFE. As the initial target, we chose MediaTek, one of the large vendors. Their chips are common in many sought-after mid-tier phones like the *Xiaomi Redmi Note 8 Pro*, with offerings from most vendors, including Motorola, Nokia, HTC, and others. The cellular baseband has close ties to the mobile operating system. Calls and data are routed through it and most of the lower-level interactions, such as establishing a call and selecting a base station, are done directly in the baseband but need to be displayed to the user.

Baseband firmware has one of the widest attack vectors of all components in modern smartphones. Every cellular network usage passes through the baseband. Countless high-complexity protocol parsers are part of the firmware. However, to this date, security testing of mobile basebands is either performed in a black box fashion or through manual static analysis. Baseband and device manufacturers are trying to limit the direct access to the baseband in various ways, such as blocking JTAG access as well as encrypting parts of or the entire baseband firmware [19, 35]. The secrecy and closed-sourceness, sometimes scrambled firmware, and usages of relatively unknown architectures, such as Qualcomm Hexagon, increase the barrier to mobile baseband analysis, while at the same time making it a more interesting and rewarding target.

In this paper we take a novel approach: by rehosting parts of a memory dump of a major cellular baseband, we are able to run main event handlers in our analysis platform. We propose *BaseSAFE*, a platform combining speedy emulation with fuzzing and heap sanitization. Building on the popular *Unicorn engine* emulator, BaseSAFE allows us to perform coverage-guided fuzzing on the MediaTek baseband. We automatically map signaling messages to their respective functions using coverage feedback, to keep false-positives low. BaseSAFE implements a custom drop-in sanitizing heap allocator for Unicorn, which can replace any baseband-internal allocation mechanism to uncover heap corruptions and use-after-frees with fuzzing.

The sample harness of BaseSAFE for LTE Radio Resource Control (RRC) [3] messages ships with thousands of unique, minimized, valid inputs for a layer 3 parser in the MediaTek baseband, the `errc_event_handler_main` function of the firmware. All samples trigger different code paths in the parser and will likely be a good set to fuzz the same parsers in other baseband firmwares. The signaling messages can be sent to the phone by a modified base station unauthenticated.

The key contributions of this paper are summarized as follows:

- BaseSAFE is an emulation platform offering zero-overhead Rust bindings for emulation, sanitizing, and fuzzing,
- The developed fuzzing toolkit can be used for any other baseband or kernel,
- We show the viability of automatic test-case inference through coverage feedback,
- We provide insights into a major baseband, MediaTek, and discuss bugs found with BaseSAFE,
- We fuzz layer 3 signaling message handlers, as some layer 3 messages are unencrypted and allow pre-authentication attacks,
- We provide a proof-of-concept use-case of BaseSAFE fuzzing RRC signaling messages and Non-Access Stratum (NAS) EMM messages in MediaTek basebands.

With BaseSAFE, we provide the groundwork for automated fuzz tests of low-level parsers in closed-source, embedded targets.

Availability

BaseSAFE is built on open-source software. Its source code and test cases are open-sourced at <https://github.com/fgsect/BaseSAFE>.

2 BACKGROUND

This paper can be seen as the intersection of two fields: cellular baseband research and fuzzing. Because of this, we will give a thorough introduction to both topics, with the goal that the reader will be able to follow the paper, independent of the background.

2.1 Fuzzing

Fuzzing is a powerful way to detect vulnerabilities, especially in low-level code. In recent years, fuzzing of desktop software and parsers using tools like *AFL* and its fork, *AFL++*, has become one of the main tools for automated analysis [26]. The fuzzer reruns the target with different input thousands of times per second, using heuristics to mutate the test cases. A key factor to a fuzzers' success is feedback from the target, usually coverage feedback. Thanks to coverage feedback the fuzzer knows if the last mutation triggered a, potentially vulnerable, path in the program [60].

Fuzzing of embedded systems is a challenging task, especially if feedback should be collected and memory corruptions should be detected quickly [37]. A few years ago, fuzzing wireless stacks, firmware or a kernel, required complex setups. They had to resemble real-world scenarios, like dedicated rogue access points [12] that are difficult to integrate into feedback-based fuzzing methodologies. In contrast to user-land software, in bare-metal systems and firmware, any state change affects the whole system. Recovering from crashes is oftentimes impossible.

2.2 Unicorn

Unicorn engine (or just *Unicorn*), a corner stone of BaseSAFE, is a fork of QEMU [41]. Unicorn extends QEMU with an easy to use API, exposing functions like reading and writing memory, and hooking specific addresses and memory accesses with custom callbacks. Unicorn offers bindings for a range of languages. However to use Rust in BaseSAFE, we extended the 3rd party *unicorn-rs* bindings, as no official Rust bindings exist [16].

Unicorn supports a vast range of processor architectures, including MediaTek's baseband architectures, ARM and MIPS [41]. This makes emulation of arbitrary code, even for embedded architectures, viable.

QEMU, as well as the forked Unicorn engine, work by performing the following steps for each new code location that needs to be run [10]:

- (1) Check if the translation block (instructions to the next conditional jump) at this location were previously cached.
- (2) If not cached, decode and lift the translation block at this address from the target platform's instruction set to *Tiny Code Generator (TCG)*, the internal intermediate representation.
- (3) Translate the *TCG* to the host platform's instruction set.
- (4) Cache the translated block.
- (5) Store a mapping from source program counter to target program counter in an address lookup table.
- (6) Execute the translated block.
- (7) Repeat for the next discovered block.

The translation blocks are similar to basic blocks by design [9]. Leveraging the correspondence between translation blocks and basic blocks, and because execution is handed back to the emulator after each run, it is possible to implement an instrumentation similar to the compile-time instrumentation, using the program counter as feedback on each new basic block. *AFL++* offers this instrumentation with *QEMU mode*. It leverages a patched version of QEMU that reports executed branches back to *AFL* [7]. After a new basic block is translated, the fork server's parent is informed that the block has been translated to ensure every block is translated only once. The control returns to the emulator after each block, which is extended with calls to `af1_maybe_log`. This call fills a shared memory section, passing the instrumentation information to *AFL*.

We built up instrumentation for BaseSAFE very similar to the *AFL-QEMU* instrumentation. Translation blocks are cached in the parent process of the SafeBASE forkserver to increase the throughput of future runs. *AFL* merely has to start the harness and generate inputs. The forked Unicorn used for BaseSAFE makes use of the same concepts, tightly.

2.3 Cellular Baseband

Every modern smartphone has multiple types of processors. Apart from the application processor, which runs the mobile operating system (OS), modern smartphones use independent baseband processors. The baseband processor handles all cellular communication. Even though the application processor and baseband processor are physically integrated into a single processor die, they are logically separate. Baseband processors run a different operating system, usually a real-time OS (RTOS), not a full-featured system like the

main processor does. Depending on the make, the method for inter-communication between both processors varies. As smartphones require faster mobile internet speeds, the internal interconnect bus requires more bandwidth, so sometimes processors also utilize shared memory between application and baseband processor. Thus, in some cases, attacking a smartphone OS via its baseband is also possible [35].

One of the main tasks of the cellular baseband is to identify the cellular networks, to authenticate, and to connect to the correct one. To achieve this, the baseband scans for the radio signal coming from the cells and decodes network information messages from *System Information Block (SIB)* messages to identify the network. After identifying nearby cells, it will connect to the strongest cell and perform the registration procedure. Once everything is done, the cellular services are available. After a connection has been established, the baseband provides mobile telephony and data services to the mobile OS.

As pointed out by Rupprecht et al. [47], various attacks are possible just with an attacker-controlled rogue base station. One of the earliest attacks targeting a baseband was presented by Weinmann [57]. Since there is no reliable way to identify whether the base station is genuine or not, an external attacker can set up a rogue base station (also known as IMSI catchers [54]) and send signaling messages like the legitimate operator, without being noticed by a user. It is possible for an attacker to inject modified signaling messages by utilizing the modified open-source software (e.g. Osmocom [43] for 2G, OpenBTS-UMTS [42] for 3G, srsLTE [20] for 4G) or the software of a commercial base station. With the wider availability of software-defined radio (SDR) devices, the total cost for the rogue base station setup became feasible to the attacker nowadays. Broadcasted signaling messages are processed by devices without explicitly establishing the connection, while dedicated signaling messages are available only after establishing a connection to the base station.

3 RELATED WORK

To the best of our knowledge, no fuzzing API for basebands exists so far. This section discusses related work in fuzzing and cellular security.

3.1 Emulator-Based Fuzzing

Different ways to use emulation for snapshot and kernel fuzzing exist. Notable examples include *TriforceAFL* [25] by Hertz and Newsham, as well as *kAFL* by Schumilo et al., both extending AFL's QEMU mode to fuzz whole virtual machines. In both cases, the fuzz driver communicates with QEMU through additional hypercalls [51]. The current state of the art kernel fuzzer for desktops, *Syzkaller*, uses VMs as well. A user-land stub inside the VM triggers the kernel via syscalls [15]. Likewise, *Unicorefuzz*, by Maier et al., fuzzes kernel functions in a QEMU-based emulator, Unicorn engine. In contrast to the other kernel fuzzers, *Unicorefuzz* only maps memory actively used in the fuzzed function and runs this single snapshot continuously [34], forwarding each newly accessed memory from the targets using *avatar*² [36]. Schumilo et al. take it one step further, fuzzing hypervisors instead of kernels, in a similar fashion inside a custom hypervisor [50].

3.2 Baseband Research

The root cause of previously proposed attacks targeting a cellular baseband can be traced back to the specification and the implementation. The errors in the standard allowed various forms of privacy leakage such as cellular subscriber tracking [52] and data eavesdropping [49]. While there had been previous works on individual baseband bugs [39, 46, 48], we are not aware of a systematized approach on identifying the baseband bugs based on fuzzing.

Features of a baseband that were proven exploitable in the past are SMS messages and AT commands. The 3GPP SMS specification [2] defines more than just a text SMS, and processing of the SMS had been exploited by previous researchers. Mulliner and Miller [40] presented the bugs related to the SMS processing of mobile basebands, and Mulliner et al. [39] tested them over-the-air for multiple types of devices. These bugs are also affecting SMS-based applications, such as one-time password [38] and SIM toolkit [8]. While mobile messenger services are replacing SMS, they are part of the cellular standard and SMS functionality exists in smartphones.

Originally designed for controlling dial-up modems, AT commands are also used in some modern smartphone basebands [1] as a part of inter-processor communication. As a result, by sending a malicious AT command it is possible to exfiltrate information from it or crash it unknowingly from the mobile operating system. Example of AT command handler fuzzing includes the work by Tian et al. [55] and *ATFuzzer* by Karim et al. [31].

Other work on fuzzing the components of smartphone includes *PeriScope* by Song et al. [53] targeting device drivers and Hay [22] targeting Android bootloaders.

The analysis of a baseband firmware is not well-researched: Qualcomm, one of the major baseband manufacturer, uses a custom in-house architecture named Hexagon. Although Qualcomm provides SDKs for the Hexagon processor, few well-known disassemblers integrate it, one notable example being GSMK's IDA Pro plugin [23]. This further limits the research using other tools. Nevertheless, most basebands use standard architecture, at least for the main, (non-DSP) processor. Golde et al. [19] and Miru [35] present a baseband firmware disassembly using industry-standard tools, for basebands using off-the-shelf architectures such as ARM and MIPS.

There had been attempts on fuzz testing cellular protocol implementations. Johansson et al. [30] proposed a cellular protocol fuzzing framework, which is integrated into the existing telecommunication testing infrastructure. Similar commercial services exist, such as P1 Telecom Fuzzer [44]. Hussain et al. proposed *LTEInspector* [27] and *5GReasoner* [28], which utilize formal analysis on the 3GPP specifications to test the implementation of basebands. Their approach is based on a formal analysis using the cellular specification, translated into a machine-readable form. *LTEFuzz* by Kim et al. [32] uses predefined test cases to identify implementation problems of a baseband. *SpikerXG* by Hernandez et al. [24] fuzz firmware of Android devices and propose an analysis platform for further rehosting and analysis will be possible, taking a big step towards automated baseband analysis. Prior publicly known baseband fuzzing setups fuzzed leaked binaries compiled for host system [35]. This is a valid approach but only feasible if object files are available, not for off-the-shelf firmware blobs.

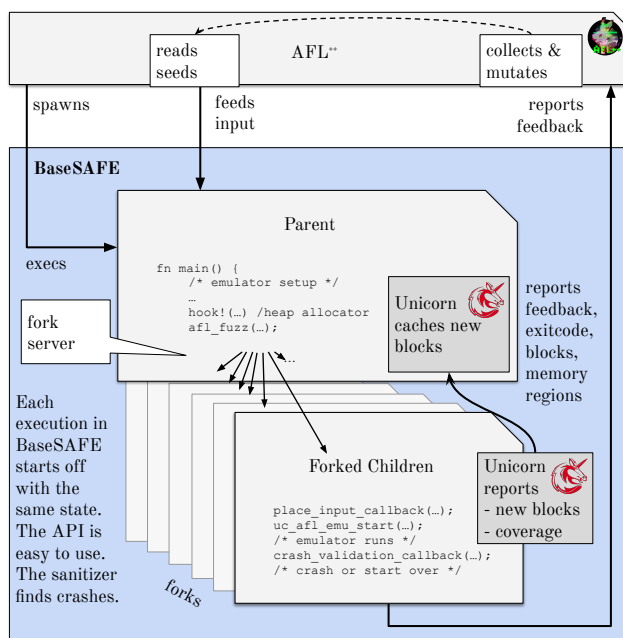


Figure 1: BaseSAFE

4 REHOSTING AND FUZZING

BaseSAFE is a platform to build baseband fuzzers on. The novel core components of BaseSAFE are an API to hook up the emulation toolkit with the maintained AFL fork AFL++ [26] in a very quick and flexible manner, as well as various useful components like a heap sanitizer. To enable high-speed fuzzing, BaseSAFE uses emulation, built upon *Unicorn engine*, a popular CPU emulator [41]. This means, fuzzing of the baseband firmware does not take place on the phone hardware, but instead interesting functions are fuzzed directly inside Unicorn engine. The Rust API makes it easy to write powerful high-performance hooks, abstracting away potential hardware interactions, interrupts, etc. This allows emulation and fuzzing of the important parts of a firmware blob on fast desktop machines. Of course, different targets do still need manual setup, coding, and reverse engineering. In the following, we will discuss the usage and benefits of BaseSAFE.

4.1 Fuzz API

In this section, we discuss the BaseSAFE API. Unicorn is extended with an AFL-specific API to enable easy fuzzing, and an extra API to hook the operating system’s heap sanitizer. A high-level overview of the BaseSAFE procedure is depicted in Fig. 1. The API of BaseSAFE goes beyond previous emulators, such as AFL Unicorn by Voss [56] which did not offer interactions with AFL. Interactions with AFL are required to kick off the fast persistent mode, but simply always started fuzzing after the first instruction. As execution left the emulator after the first instruction to read AFL input, all Unicorn translation caches were flushed constantly. On top, it only worked for harnesses written in the interpreted and garbage collected language Python.

The Unicorn engine API includes functions to set page mappings, read and write memory and registers, add hooks, as well as start and stop execution with different conditions. BaseSAFE makes them available via Rust.

We extended this part of BaseSAFE with the following methods, tailored for fuzzing:

`afl_forkserver_start`. After the initial setup of the test case is done, the fuzz harness can call `afl_forkserver_start`. This kicks off the forking logic. The baseband is kept in the same state in the parent process, each fuzz test case is executed against a forked copy of the emulator. So a call to `afl_forkserver_start` effectively freezes the current state of prior to fuzzing run and at the same time tells the attached `afl-fuzz` process to generate inputs. After the forking started, the harness should read the input for this run from `AFL++` and place it into the appropriate location in the target's memory. As the fork is copy on write, the test case data will be reset at the end of the run. Once this happens, the parent process requests the next test case. Furthermore, the forking contains a caching mechanism for Unicorn's JIT, inspired by the `AFL QEMU` mode: for each uncached basic block the child encounters, the child will

- (1) Decode the basic block from the target architecture (for MTK, this is either 32 bit ARM or MIPS).
- (2) Add instrumentation to the block, namely register each jump from one location to the next in a shared map that will be evaluated by AFL++ to generate further inputs.
- (3) Notify the parent process about the current block address and flags.
- (4) Run the basic block.
- (5) Continue with the next block. If it is already cached, patch in a direct jump.

Once notified about a new block address by the child, the parent process will also translate this block. As the parent mirrors the child’s translation, this block will already be present in the block cache for the next test case. This concept was carried over from AFL’s QEMU mode, albeit with speed improvements: whereas QEMU mode caches inside of the emulator, BaseSAFE patches it into the translated block itself, reducing the need for indirect jumps, a method first proposed by Biondo [11].

`afl_next`. As the fork syscall is heavyweight and, therefore, rather slow, BaseSAFE offers a faster alternative: persistent mode. For targets like single parsers, for example `errc_event_handler_main`, afl’s persistent mode can greatly improve fuzzing speeds. Instead of exiting and reforking, the child process resets its state internally, resets the needed stack, memory, and registers, and then calls `afl_next` to inform the BaseSAFE parent and afl-fuzz about the end of a single fuzz run. AFL++ will then place the next test case and the child can call the fuzzed target again.

`af1_emu_start`. In contrast to the `emu_start` function offered by the core Unicorn emulator, `af1_emu_start` of BaseSAFE takes multiple exit addresses. This is required if the target may not always return at the end of a function, for example if error conditions trigger. On top, it will not clear the translation block cache, containing the JITted basic blocks after execution, as it is the case with the emulator function in Unicorn. This allows us to reuse the cache for

consecutive forks, as discussed in `afl_forkserver_start`, as well as persistent mode, discussed in `afl_next`. Recompilation of the basic blocks is therefore not needed. Fuzzing speeds are high once the emulator has encountered and translated most of the blocks.

`afl_fuzz`. Instead of manually specifying the fuzzing logic, such as reading AFL’s input, catching Unicorn exceptions, or looping back to the beginning for persistent mode, the developer may choose to use the all-in-one function `afl_fuzz`. After setting up the baseband inside the emulator, the fuzz function takes over all necessary steps to fuzz, including all of the functions mentioned above. Its signature can be seen in Listing 1.

The function `afl_fuzz`

- (1) Loads the current test case input from AFL.
- (2) Calls the `place_input_callback`, in which the harness should write the input into the emulator memory at the appropriate position. For persistent mode, the emulator has to reset additional state changes in this step.
- (3) Runs the emulator until execution reaches one of the exits, a hook crashes execution or an illegal state occurs in the emulator.
- (4) Checks the Unicorn emulator error conditions and (optionally) calls the `crash_validation_callback`, allowing us to implement custom sanitization routines.
- (5) Starts from the top if persistent mode is enabled and the counter did not expire.

```
pub fn afl_fuzz <F: 'static, G: 'static> (&mut self,
    input_file: &str,
    input_placement_callback: F,
    exits: &[u64],
    crash_validation_callback: G,
    always_validate: bool,
    persistent_iters: u32)
    -> Result<(), AflRet>
where
    F: FnMut(UnicornHandle<D>,
        &[u8], i32) -> bool,
    G: FnMut(UnicornHandle<D>, uc_error,
        &[u8], i32) -> bool {
```

Listing 1: Function Signature of `afl_fuzz` in Rust.

In the `input_placement_callback` the harness writes the input test case to the emulator memory. The callbacks both get pointers to the input of the current test case, provided by AFL, as well as the persistent iteration index if it is required. Furthermore, the `crash_validation_callback` will get the exit code from Unicorn, which will indicate errors caught during emulation, such as out-of-bounds memory accesses. Depending on its return code, the exit condition may be considered interesting, e.g., a crash or hitting sanitization, in which case the AFL process gets the information forwarded over an Inter-Process Communication (IPC) mechanism. The fuzz function also takes a list of exits at which emulation will stop, a flag whether the validation callback should also be called without a Unicorn error condition and an additional u32 counter, indicating if—and how often—persistent mode should loop before forking again.

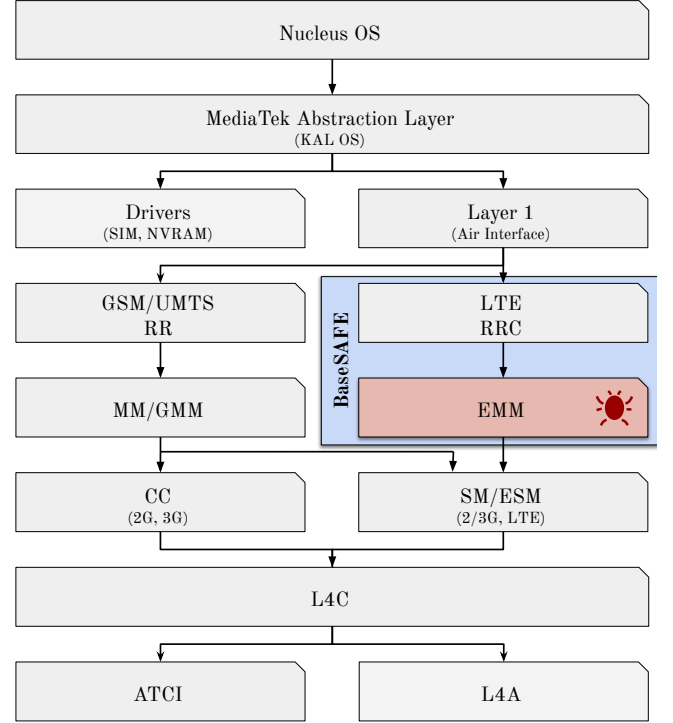


Figure 2: MediaTek Baseband Architecture.

Debug Tracing. While execution has to be as fast as possible for fuzzing to allow a large amount of test cases to be evaluated, requirements when triaging a crash are vastly different. Instead of high speed, the user wants a good understanding of the execution path taken and encountered error cases, and hence needs as much context as possible. For this, BaseSAFE features a debug mode that will output all disassembled instructions and register values during execution. As long as debug outputs are enabled, our harness also prints all output directly generated by the baseband, such as logs generated by `dhl_trace` and assert messages.

4.2 The MediaTek Baseband

We chose a MediaTek baseband as a target to implement a proof-of-concept fuzzing harness, proving the usability and viability of BaseSAFE. First, we will present an overview over the reverse-engineered baseband.

For the course of this work, we focus on the *HTC One E9+*, using MediaTek’s Helio X10 (MT6795) processor launched in 2015 [58]. Aside from HTC One E9+ and M9+, this processor is also used in some smartphones around 2015, including Sony Xperia M5 and Xiaomi Redmi Note 2 and 3. We were able to access the unencrypted baseband firmware using the tool from Miru [35], which is targeting the same processor but tested in a different device. MediaTek’s baseband firmware consists of two parts: ARM and DSP. The DSP firmware controls the lower layers, including modulation and demodulation of the over-the-air signal. The ARM firmware controls the upper layers, including the processing of the signaling messages

and interconnection with the application processor. We are focusing on the ARM part of the baseband firmware, as this is the place where cellular control plane messages are processed and, therefore, can be controlled by an external attacker.

An overview of the internal structure of the ARM firmware is depicted in Fig. 2. Most of this MediaTek *MTK* baseband OS was likely carried over from their prior feature phone OS, providing the same task model and IPC mechanisms [33]. The OS is generally following the layer model of the cellular network. The lowest layer is a Nucleus RTOS kernel with a MediaTek abstraction layer above it. Drivers are interacting with external entities used by a baseband, such as NVRAM, used to store baseband configurations, and SIM cards. Layer 1 and 2 are not clearly separated within the MediaTek baseband firmware. Layer 3 consists of multiple protocols related to the separate functionalities of the cellular network: radio resource control (RR, RRC), mobility management (MM, GMM, EMM), call control (CC), and session management (SM, ESM). Above this lies an application layer, layer 4, whose implementation is MediaTek-specific. It consists of a command interpreter (ATCI), a control entity (L4C) and an adaption layer (L4A) and interacts with the mobile OS running on the device's application processor.

We are focusing on LTE RRC and EMM messages, as part of these messages are normally used for identifying a base station and exchanged upon establishing a connection between smartphone and network. As such, vulnerabilities in handling these messages enable a pre-authentication attack.

Imagination Technologies—then owner of MIPS—announced that MediaTek is adopting MIPS as the architecture for their baseband [29]. This is a departure from the ARM-based core that we analyzed as a proof-of-concept for this work. By analyzing the smartphone's baseband firmware images by chipset using binwalk opcode analysis and Ghidra, we also found that MediaTek used ARM in their basebands released before 2017 (Helio P25, X10, X27) and switched to MIPS around mid-2017 (Helio P23, P70, P90, X30). Our reverse engineering indicates that the parsers we are fuzzing are close to identical in MIPS-based MediaTek basebands. This is to be expected, as code reuse across architectures minimizes the development costs. Of course, the proof-of-concept harness of BaseSAFE can easily be ported to MIPS, as Unicorn itself supports the architecture, although it might not provide novel insights due to the aforementioned code reuse. Likewise, the MTK toolkits are rather standardized for both ARM and MIPS.

4.3 Nucleus and MTK Firmware IPC

The ARM firmware we used as an example application for BaseSAFE is based on the Nucleus RTOS kernel, the libraries indicate a kernel version of 2.x. The Nucleus RTOS is a non-preemptive realtime OS with a queue-based IPC mechanism. The main method for the different parts of the MTK firmware to communicate with each other and forward packages to higher layers are these queues. Modules can use the `do_send_msg` function or its wrappers to send so-called Inter Layer Messages (ILMs) to the internal or external queue. The function proceeds to call `kal_enqueue_msg` in the Kernel Abstraction Layer, which uses Nucleus primitives to pass the MTK message representation to the Queue Management Unit (QMU) of the kernel. On the other end, modules use `msg_receive_extq` or

`msg_receive_intq`, wrapping `kal_dequeue_msg`, in order to receive ILMs for further processing. Each queue item, i.e. ILM, contains an identifying message ID and destination module ID. The MTK firmware uses this information tuple to route the queue entry internally and start the correct handler. Message IDs then provide hints on how to process messages, e.g., parsers handling incoming (physically external) messages could be informed on which decoder to use. Additionally, ILMs handle domain-specific manifestations of local parameters and peer buffers, carrying various forms of information. When an ILM is initially constructed, the functions `construct_int_peer_buff` and `construct_int_local_para` can be used to allocate and populate the respective buffer. As a next step, the `get_int_ctrl_buffer` wrapper calls into `kal_get_buffer` which communicates with the Nucleus Partition Manager to return a fresh memory chunk. Both buffers have reference counts, a module currently using one of them would signal its demand by calling e.g. `hold_local_para` and thus increasing the counter. Specific free wrapper functions, namely `free_int_local_para` and `free_int_peer_buff`, first check the reference count and free the corresponding buffer only if it is 0, otherwise it is decremented by 1. The memory layouts of the ILM and local parameter structures are depicted in the upper part of Fig. 4. The MTK firmware also implements an own queue management unit for the buffer management of incoming messages and wraps received messages in metadata, as depicted in the diagram in the lower part of Fig. 4.

4.4 Selective Emulation

Instead of emulating the whole baseband, we selectively emulate single parsers. The selective emulation is single-threaded and only spawns one process. This allows us to emulate quickly and with zero false-positives. However, this way we have to be more selective about the portion of the baseband we fuzz—as manual effort is required for each harness.

As illustrated in Fig. 2, we are focusing on layer 3 control plane signaling messages, in particular, 3G [6] and LTE [3] RRC and NAS EMM [4, 5] messages. They are relatively easy to modify with a fake base station compared to lower layers. Some of these messages, namely SIBs and *Paging* messages are not encrypted and decoded automatically upon knowing the cell even without connecting to them. Even though these messages have been a target for various previous works, they still remain valid as a fuzzing target for various reasons. Signaling messages are encoded in a binary-based format such as ASN.1 and CSN.1, and correctly implementing a parser for those protocols can be bug-ridden, while modern ASN parsers are oftentimes autogenerated, making these parts less of a pressing issue. Even though there are compliance tests for the standards, these are developed towards the interoperability among multiple vendors, not towards the implementation problems of an individual baseband. The large amount of more than one thousand different signaling messages for LTE RRC alone we were able to deduce through fuzzing is likely not completely covered by compliance tests. Hence, it is likely that individual baseband manufacturers can make individual mistakes, which may not have been filtered by their internal testing.

To this end, we identify the handling functions of the aforementioned signaling messages of the baseband firmware structure with

minimal user input. We start from the downlink RRC messages of the cellular network, which is categorized as follows:

- PCCH: Delivers paging messages. Plaintext message without authentication.
- BCCH DL-SCH: Delivers SIBs to identify the network. Plaintext message without authentication, baseband automatically receives the contents upon cell discovery.
- CCCH: Initiates a connection between the phone and the base station. Plaintext message without authentication.
- DCCH: Dedicated channel between the phone and the base station. Integrity protected and optionally (but usually) encrypted.

In addition to RRC messages, NAS EMM messages manage the registration of a phone and mobility. Some of the messages are exchanged without encryption, therefore malicious EMM messages could be used as pre-authentication attack.

We have collected signaling messages from the real network and phone using SCAT [45] and use them as seed inputs to potential signaling message handlers.

The different decoders expect input buffers to be placed at specific offsets in multiple layers of internal structures. For incoming signaling messages, new buffers are allocated by the MTK-internal QMU. These buffers contain administrative queue metadata as well as the size of the incoming message and a pointer to the payload buffer. A local parameter struct is populated with a pointer to the queue buffer and is being held by an Inter Layer Message struct. Finally, the ILM needs to contain the correct message ID, signaling the higher-level function into which ASN.1 decoder the incoming message should be passed. Fig. 4 depicts the whole layout using a PCCH message as an example. Correct placement of buffers, pointers and length fields can be handled in the `place_input_callback` discussed in Sect. 4.1.

4.5 Parser Deduction

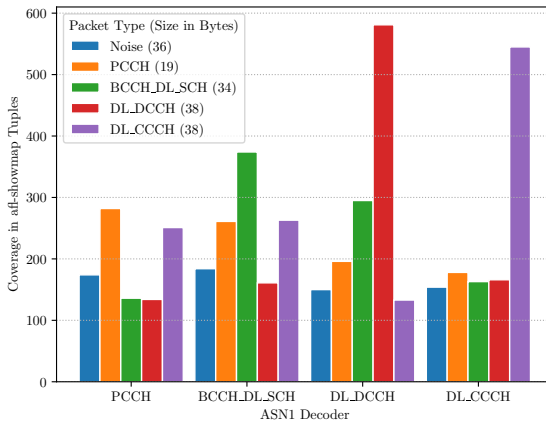


Figure 3: Reported edge coverage of packages for each parser. The correct packages reaches the highest coverage in each.

BaseSAFE can guide reverse engineers to select good fuzzing targets with a concept that we call *parser deduction*. After we collect signaling messages we are interested in using SCAT, we deduct the correct parser for the signaling message by evaluating the code coverage of different input functions when presented with the signaling message, compared with code coverage of unrelated packages. For this, we feed valid signaling messages into all decoders by emulating valid IPC messages and log the coverage. As shown in Fig. 3, the correct signaling message always reaches the highest coverage in the correct parser, as expected. The correlation for PCCH is smaller than that of the other signaling messages, likely due to the fact that the valid PCCH message we used was rather small, with only 19 bytes. To introduce a baseline, 30 noise signaling messages with 32 to 42 random bytes and an average length of 36 were generated. These were fed into the discussed decoders and achieve notably less coverage than the valid signaling message for each decoder, the average coverage can be seen in Fig. 3. This method of parser deduction allows us to select an interesting and correct target to selectively emulate and fuzz.

4.6 Rehosting the Baseband with Rust



Figure 4: Structure of MTK ILM, encapsulating a PCCH Message including pointers needed for correct input placement.

Fuzzing speed depends on various factors. Instrumenting a binary using a debugger degrades performance. Recent advancements focus on speed of path finding, but also on sheer execution speed of the instrumentation through lightweight hardware features [51, 61], by leveraging optimized runtimes like QEMU block chaining [11] or vectorized virtualization [17]. Faster instrumentation could be possible.

Current research has shown that AFL’s QEMU mode’s performance can be improved by re-enabling QEMU’s block chaining, which merges code blocks if one ends with a direct jump. It is disabled because it interferes with AFL’s instrumentation: Merged

blocks don't jump back into the emulator after every single contained block, so it effectively disables tracing direct jumps. The author injects the instrumentation code into the translated code, and thus can safely enable block chaining. Combined with proper caching this yields a speedup of 3-4 times the mainline QEMU mode [11]. This patch could be ported to AFL-Unicorn, and could significantly reduce the performance gap to compiler-assisted instrumentation.

4.7 Drop-In Heap Sanitizer

Muench et al. classify the results of memory corruptions found through fuzzing in different categories. In their book, only *observable crashes* and *hangs* are easy to track. Especially in a fuzzing scenario, where the same test-case restarts over and over, the categories they classify as *late crashes* and *malfunctioning* are impossible to spot. Of course, the last class, *no effect*, is impossible to track down without address sanitization [37]. A memory corruption itself and the subsequent use of the corrupted memory may often be far apart. At this point the fuzzer already stopped the execution of this run and started the next iteration, removing all traces of the bug. Thus, even when a crashing input is given, detecting the underlying memory bug is impossible. We solve this issue by implementing a drop-in allocator that makes use of the emulator features to provide sanitization.

4.7.1 How to Drop-In. Usually, a drop-in allocator would be put in place, either during compilation or linked dynamically when the program is loaded. Both options are not applicable for our use case: embedded firmware without source code, tool-chain, or knowledge about the linker. While binary patching could be considered, similar to RetroWrite by Dinesh et al. [14], adding functionality in the emulation layer is less fragile and works for all supported instruction sets. The hooking functionality of the Unicorn engine inserts checks for conditions and callbacks directly into the JITted code.

In contrast to the QASan sanitizer for QEMU by Fioraldi, that patches each memory access in QEMU [18], we add the instrumentation on top of the memory access hooks already offered by Unicorn. With a hook in place, Unicorn engine emits checks for conditions like memory accesses and executed instructions. If the check triggers, the placed hook is called from the JITted code directly, without stopping the emulation. Using this feature, we overwrite the firmware's internal allocator, i.e. `kal_get_buffer` described in Sect. 4.3, by hooking its address. Whenever the hook triggers, we allocate memory in a previously mapped page and pass the location to the firmware by filling in the correct register. After the hooked allocation, we increase the program counter to skip the firmware's actual allocator function call.

4.7.2 Allocator Implementation. The custom allocator offered by BaseSAFE itself makes heavy use of Unicorn hooks for sanitization. Before emulation starts, a memory region large enough to handle all possible allocations during a single run gets allocated inside the emulated target. Initially, an access hook is placed on the whole region. Each time the hook triggers, a memory out-of-bounds access is detected. The nature of the hook allows us to distinguish between reads and writes and allows us to log the current instruction pointer.

Allocation. When the baseband's allocator would be called, the Unicorn hook calls our allocator instead. The allocation chunk is allocated by removing our memory hook from a portion of the memory region of this size. A canary region with varying size, depending on the size of the allocation, is left hooked between each allocated chunk. The canary region is, again, protected by hooks. This way the only heap corruptions we cannot spot are writes skipping the variable-sized canary regions and accessing another already allocated chunk. Normally, all out-of-bounds accesses are detected.

Deallocation. The deallocator hook places a new memory access hook on the previously allocated region. Since the memory region will never be reused for this single run, all use-after-free bugs are detected by this hook. In addition, the chunk size, being part of a bookkeeping structure, gets set to 0. Whenever free is being called on a chunk with a size of 0, a double-free is detected. The managing structures are not part of the heap itself and are kept outside of the emulator and hence cannot be impacted by memory corruptions. As the forkserver will reset the emulator memory for each emulation pass by reforking, we do not have to clean the custom heap manually, unless the persistent mode is in use, in which case the hooks are replaced completely.

5 EVALUATION

In the following, we will first discuss results gathered with BaseSAFE, including memory corruptions. Then, we discuss how BaseSAFE repackages PCAPs, and how we replay the test cases against a real MediaTek-based device over-the-air.

5.1 Exhaustive Test Cases From LTE RRC

We ran BaseSAFE on the LTE RRC test case for around one week with about 15k executions per second. Benchmarking the fuzz case on an Intel i7-6700 CPU @3.40GHz, the speed fluctuated around 1.5k executions per second on a single core. Although no crashes could be found during the RRC tests, we were able to uncover slow paths. However, this cannot be abused for denial-of-service because LTE RRC timers [3] reset the internal parser if signaling messages were not received within the time limit.

In total, after minimizing millions of test cases using *afl-cmin* on the corpus and *afl-tmin* on every single corpus-minimized input, 1388 unique test cases, leading to unique paths in the parser, were found, see Sect. 5.3. They will be released as part of BaseSAFE and can be reused to test the same message on other baseband firmware in the future. This number can still go up with longer fuzzing times. As these signaling messages were effectively outgenerated by the parser, all of the messages are relevant for parsing, even if some behavior may not be specification-compliant, cf. Fig. 5.

5.2 Memory Corruptions in NAS EMM

Higher up in the LTE stack, in the NAS EMM parser, BaseSAFE was able to uncover out-of-bounds reads and writes. The position of the bug within the MTK architecture is shown in Fig. 2. After fuzzing the parser, the heap sanitizer, discussed in Sect. 4.7, successfully reported multiple such crashes with the same root cause. The bug, first discovered by Grassi and Chen [21], lies in `decodeEmergencyNumberList`, a function being called during the

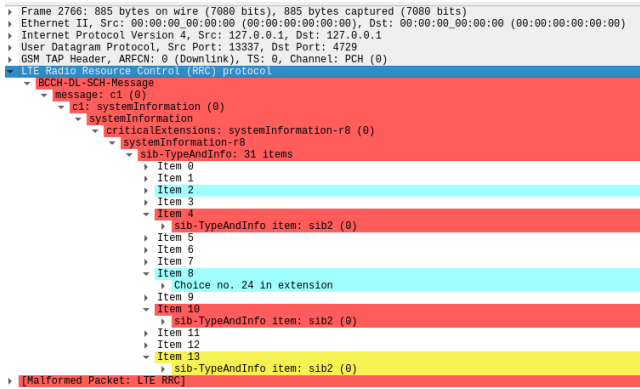


Figure 5: Wireshark trace of one of the fuzzer-generated signaling messages triggering unique code flow in MTK

handling of *Tracking Area Update Accept* and *Attach Accept* messages, specified in the 3GPP TS 24.008 [4] standard. Here, an attacker-controlled length-byte is not checked against the correct message size, leading to an out-of-bounds *read* from the received message buffer while copying it to the internal data structure. As the offset into the target buffer gradually increases by factor $0x2b$, an attacker would in addition be able to overflow it during the copying process and achieve an out-of-bounds *write* on the heap.

```
i = 0;
do {
    index = *ecc_number_list_struct;
    curr_item = ecc_number_list_struct + (uint)index * 0x2b;
    curr_item[2] = msg[i] - 1;
    data_start = i + 2;
    curr_item[1] = msg[i + 1] & 0xffff;
    j = 0;
    while (i = data_start & 0xffff, j < curr_item[2]) {
        data_start = i + 1;
        curr_item[j + 3] = msg[i];
        j = j + 1 & 0xff;
    }
    j = (uint)index + 1 & 0xff;
    *ecc_number_list_struct = (byte)j;
} while (i < length);
dhl_trace(TRACE_GROUP_1, 0, DAT_001b8020, PTR_DAT_001b8024, j, length, msg_ptr);
```

Listing 2: Loop in decodeEmergencyNumberList

5.3 The Red Pill

To finalize the emulation evaluation, we need proof that the messages produced in the emulator are indeed valid messages for real basebands. For this, we used two main methods.

5.3.1 AFL Inputs to PCAPs. After running for five days, the feedback-driven mutations of AFL generated a large corpus of potential inputs, with over 250k queued items. Of course, many of these are rather similar. As BaseSAFE works together with all AFL++ tools, we can use a combination of AFL’s minimization tools to reduce the amount of test-cases to 1388+ unique tests. To arrive at this number:

- (1) We ran `afl-cmin`, which loads the coverage map for each test case, then removes all test cases that only touch the same code paths from the list, keeping the smallest for each. This removes all files that do not reach new code paths, making sure each test case is actually relevant for the parser.



Figure 6: Base station setup with our test phone HTC One E9.

- (2) We ran `afl-tmin` multi-processed on all remaining files. The test case minimization overwrites random chunks of the input file. If the coverage map stays the same, parts of the chunk are removed to check if the map also stays unchanged without these bytes. Similar heuristics are repeated until the file size is as minimal as possible—while maintaining the coverage.

After the minimization process, we are left with a minimal set of inputs that still cover all possible branches of the original baseband parser. In contrast to official test-cases, they may not be valid packages—but they will still trigger new conditions in the parser. See, for example, the dissected packet containing signaling message in Fig. 5. To arrive at this dissection, and verify our method, we wrap the minimized test cases in a valid PCAP. For this, BaseSAFE ships with a custom tool to wrap the test cases into a PCAP file. The tool writes PCAP headers and then wraps the bytes each minimized test case into a GSMTAP packet. The wrapped GSMTAP packets are decodable as a signaling message in Wireshark. One of the generated test case decoded in Wireshark is presented in Fig. 5.

5.3.2 Replay Against Real Phones. In order to try out responses in the real world and vet the behavior of BaseSAFE, we built a setup able to replay the signaling messages. For this, we put up a rogue base station capable of injecting the messages at the correct time during the connection establishment procedure. We are using a software-defined radio and freely available application OpenLTE [59], illustrated in Fig. 6. Along with this, we were able to analyze the baseband log output with the *MTKLogger* system application. The application is available on most MediaTek-based phones, although it is hidden from the user interface and needs to be accessed using special methods.

6 FUTURE WORK

BaseSAFE is an important first step towards fully automated vulnerability discovery on cellular basebands. However, it only manages to cover a small portion, leaving many areas to explore further.

6.1 Fuzzing for Logic Bugs

Currently, our proof-of-concept harness for BaseSAFE mostly detects memory corruptions during parsing, except for the places

where asserts were explicitly inserted by MediaTek into the baseband. It will not discover any other bugs. Through the insertion of additional hooks and more elaborately modeled execution flows, a variety of other bugs could be in the future. For example, finding traces that disable a timer could lead to sustainable DoS and easier exploitation of bugs [19]. On top, the parsed message structs, resulting from our LTE RRC fuzz test, could be handed to the consuming functions behind the baseband. This could yield further bugs, as packages might pass the parser without corruptions, but the consumer could, in turn, blindly trust that input, just as we saw in Sect. 5.2. Using BaseSAFE hooks, a lot of other logic bugs could also be modeled. For the baseband authentication, paths that reach an authenticated state without passing the necessary authentication functions can be hunted down this way.

6.2 Collision-Free Coverage Tracing

Right now, BaseSAFE uses the instrumentation of AFL QEMU mode. For each new translation block, it calculates a shift and a XOR operation to find the `afl_idx` offset, see Listing 3.

```
afl_idx = cur_loc ^ uc->afl_prev_loc
INC_AFL_AREA(afl_idx);
uc->afl_prev_loc = cur_loc >> 1
```

Listing 3: Instrumentation in BaseSAFE

At the position of `afl_idx`, `INC_AFL_AREA` then increases a counter at the shared map used to report feedback to AFL++.

While this hash is good enough to find almost all paths, our tests indicate that collisions occasionally occur. This leaves a small number of branches undetected, negatively affecting feedback-based mutations. The fuzzer will still eventually reach colliding paths, but may classify the edge as already taken, putting less weight on this test case. In the future, BaseSAFE will be extended with collision-free instrumentation. Initial tests without AFL’s hashing indicate a higher total number of total paths found.

6.3 Additional Targets

In the course of this paper, we were merely able to highlight a small portion of a whole operating system. The analyzed MediaTek firmware for the HTC One in question has 56 calls to `msg_receive_extq`, the IPC mechanism to receive messages from the system queue, alone. As described in 4.3, each of these calls is one specific task. Each task may contain parsers for multiple different network packets and call one of them, depending on the ILM `msg_id` (see Fig. 4 for one ILM example). Each such queue read may contain, or depend on, user-provided input and can be an interesting target to fuzz in itself. Especially 2G functions might be another interesting target as it is usually old code that may be seldomly used and tested. Apart from MediaTek, other cellular basebands, base stations, and even unrelated firmware can be tested with BaseSAFE after some adaptation. Piece by piece, support for non-standard architectures, such as Qualcomm’s Hexagon, can be added by porting their existing open-source QEMU versions [13] to Unicorn engine, thereby improving the number of testable basebands.

6.4 Further Harness Automation

While the methods and tools presented in this paper will be applicable to all cellular basebands, hook creation could be further automated to allow adaption to new platforms with less manual work. Of course, this will need a deeper automated understanding of unknown firmware blobs, using heuristics and automated static analysis.

7 CONCLUSION

BaseSAFE shows good results. It provides fast fuzzing speeds, with around 15000 executions on a small server for our real-world use-case, emulating multiple Mediatek baseband parsers. This proves that emulation is a good fit for automated bug discovery in baseband firmware analysis. BaseSAFE’s snapshot-based fuzzing introduces a new level of precision and achieves high coverage and execution speeds, not achievable by classical over-the-air fuzzing approaches. BaseSAFE is built on top of state-of-the-art open-source tools and has—in turn—been open-sourced. The API offered by its Rust bindings makes it easy to quickly implement fuzz cases without additional overhead. We were able to run high-performance partial emulations of complex firmware. The introduced drop-in sanitizing allocator finds memory corruption bugs automatically, solving a problem considered hard in prior literature [37]. Our proposed framework found vulnerabilities that are reproducible over-the-air. We conclude, that with BaseSAFE, fuzzing of embedded firmware in general, and different parts of the MediaTek cellular baseband in particular, is greatly facilitated and allows for efficient automated bug discovery in various scenarios.

Responsible disclosure. All results of this research were communicated to MediaTek in a timely fashion.

Acknowledgements

The authors would like to thank Jiska Classen and Altaf Shaik for valuable feedback.

REFERENCES

- [1] 3GPP. 2018. *AT command set for User Equipment (UE)*. Technical Specification (TS) 27.007. 3rd Generation Partnership Project (3GPP). <http://www.3gpp.org/DynaReport/27007.htm> Version 15.4.0.
- [2] 3GPP. 2018. *Technical realization of the Short Message Service (SMS)*. Technical Specification (TS) 23.040. 3rd Generation Partnership Project (3GPP). <http://www.3gpp.org/DynaReport/23040.htm>
- [3] 3GPP. 2020. *Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification*. Technical Specification (TS) 36.331. 3rd Generation Partnership Project (3GPP). <http://www.3gpp.org/DynaReport/36331.htm>
- [4] 3GPP. 2020. *Mobile radio interface Layer 3 specification; Core network protocols; Stage 3*. Technical Specification (TS) 24.008. 3rd Generation Partnership Project (3GPP). <http://www.3gpp.org/DynaReport/24008.htm>
- [5] 3GPP. 2020. *Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3*. Technical Specification (TS) 24.301. 3rd Generation Partnership Project (3GPP). <http://www.3gpp.org/DynaReport/24301.htm>
- [6] 3GPP. 2020. *Radio Resource Control (RRC); Protocol specification*. Technical Specification (TS) 25.331. 3rd Generation Partnership Project (3GPP). <http://www.3gpp.org/DynaReport/25331.htm>
- [7] AFL. 2020. *AFL QEMU Mode*. https://github.com/mirrorer/afl/blob/master/qemu_mode/README.qemu
- [8] Bogdan Alecu. 2013. SMS fuzzing—SIM toolkit attack. *DEF CON 21* (2013).
- [9] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.
- [10] Fabrice Bellard. 2020. *Tiny Code Generator*. <https://git.qemu.org/?p=qemu.git;a=blob;lain=f=tcg/README;hb=HEAD>

- [11] Andrea Biondo. 2018. Improving AFL's QEMU mode performance. *0x41414141 in ?? ()* (Sep 2018). <https://abiondo.me/2018/09/21/improving-afl-qemu-mode>
- [12] Laurent Butti and Julien Tinnés. 2008. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology* 4, 1 (2008), 25–37.
- [13] Comsecuris. 2020. QEMU with support for QDSP6 user mode emulation. <https://github.com/Comsecuris/qemu-hexagon>
- [14] S. Dinesh S. Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE S&P 2020*.
- [15] David Drysdale. 2016. Coverage-guided kernel fuzzing with syzkaller. <https://lwn.net/Articles/677764/>
- [16] SÅbastien Duquette. 2020. Rust bindings for the unicorn CPU emulator. <https://github.com/ekse/unicorn-rs>
- [17] Brandon Falk. 2018. Vectorized Emulation: Hardware accelerated taint tracking at 2 trillion instructions per second. https://gamozolabs.github.io/fuzzing/2018/10/14/vectorized_emulation.html [Online; accessed 11. Nov. 2018].
- [18] Andrea Fioraldi. 2019. Sanitized Emulation with QASan. <https://andreaforaldi.github.io/articles/2019/12/20/sanitized-emulation-with-qasan.html>
- [19] Nico Golde and Daniel Komaromy. 2016. Breaking Band: reverse engineering and exploiting the shannon baseband. <https://comsecuris.com/slides/recon2016-breakingand.pdf>
- [20] Ismael Gomez-Miguel, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Douglas J. Leith. 2016. srsLTE: An Open-Source Platform for LTE Evolution and Experimentation. *CoRR* abs/1602.04629 (2016). <http://arxiv.org/abs/1602.04629>
- [21] Marco Grassi and Xingyu Chen. 2020. Exploring the MediaTek Baseband. In *OffensiveCon*.
- [22] Røee Hay. 2017. fastboot OEM vuln: Android bootloader vulnerabilities in vendor customizations. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*.
- [23] Willem Hengeveld. 2013. IDA processor module for the hexagon (QDSP6) processor. <https://github.com/gsmk/hexagon>
- [24] Grant Hernandez and Kevin R. B. Butler. 2019. Basebads: Automated Security Analysis of Baseband Firmware: Poster. (2019), 318Å\$319. <https://doi.org/10.1145/3317549.3326310>
- [25] J Hertz and T Newsham. 2016. Project triforce: Run afl on everything. *NCC Group, Tech. Rep.* (2016).
- [26] Marc Heuse, Heiko Eißfeld, Andrea Fioraldi, and Dominik Maier. 2020. american fuzzy lop plus plus (afl++). GitHub. <https://github.com/vanhauser-thc/AFLplusplus>
- [27] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. 2018. LTEInspector: A systematic approach for adversarial testing of 4G LTE. In *Network and Distributed Systems Security (NDSS) Symposium 2018*.
- [28] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 2019. 5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11–15, 2019*, Lorenzo Cavallaro, Johannes Kinder, Xiaofeng Wang, and Jonathan Katz (Eds.). ACM, 669–684. <https://doi.org/10.1145/3319535.3354263>
- [29] Imagination Technologies. 2017. MediaTek selects MIPS for LTE modems. <https://www.mips.com/press/mediatek-selects-mips-for-lte-modems/>
- [30] W. Johansson, M. Svensson, U. E. Larson, M. Almgren, and V. Gulisano. 2014. T-Fuzz: Model-Based Fuzzing for Robustness Testing of Telecommunication Protocols. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. 323–332. <https://doi.org/10.1109/ICST.2014.45>
- [31] Imtiaz Karim, Fabrizio Cicala, Syed Hussain, Omar Chowdhury, and Elisa Bertino. 2019. Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones. 529–543. <https://doi.org/10.1145/3359789.3359833>
- [32] Hongil Kim, Jiho Lee, Lee Eunkyu, and Yongdae Kim. 2019. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1153–1168. <https://doi.org/10.1109/SP.2019.00038>
- [33] X. Lu, J. He, and J. Li. 2011. A Tibetan input method based on MTK for mobile phone. In *2011 International Conference on Consumer Electronics, Communications and Networks (CECNet)*. 3884–3887. <https://doi.org/10.1109/CECNET.2011.5768296>
- [34] Dominik Maier, Benedikt Radtke, and Bastian Harren. 2019. Unicornfuzz: On the Viability of Emulation for KernelSpace Fuzzing. In *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12–13, 2019*, Alex Gantman and Clémentine Maurice (Eds.). USENIX Association. <https://www.usenix.org/conference/woot19/presentation/maier>
- [35] GyÅrgrgy Miru. 2017. Path of Least Resistance: Cellular Baseband to Application Processor Escalation on Mediatek Devices. https://comsecuris.com/blog/posts/path_of_least_resistance/
- [36] Marius Muench, Aurélien Francillon, and Davide Balzarotti. 2018. AvatarÅs: A multi-target orchestration platform. In *BAR 2018, Workshop on Binary Analysis Research, colocated with NDSS Symposium, 18 February 2018, San Diego, USA*. San Diego, UNITED STATES. <http://www.eurecom.fr/publication/5437>
- [37] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings 2018 Network and Distributed System Security Symposium, San Diego, CA*.
- [38] Collin Mulliner, Ravishankar Borgaonkar, Patrick Stewin, and Jean-Pierre Seifert. 2013. SMS-Based One-Time Passwords: Attacks and Defense. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Konrad Rieck, Patrick Stewin, and Jean-Pierre Seifert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–159.
- [39] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of Death: from analyzing to attacking mobile phones on a large scale. *USENIX Security* (2011). http://static.usenix.org/events/sec11/tech/full_papers/Mulliner.pdf
- [40] Collin Mulliner and Charlie Miller. 2009. Fuzzing the Phone in your Phone. *Black Hat USA 2009* (2009). <https://www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf>
- [41] Anh Quynh Ngyuen and Hoang Vu Dang. 2020. Unicorn: Next Generation CPU Emulator Framework. <http://www.unicorn-engine.org/BHUSA2015-unicorn.pdf>
- [42] OpenBTS. 2020. OpenBTS-UMTS. <http://openbts.org/w/index.php?title=OpenBTS-UMTS>
- [43] Osmocom Project. 2020. Cellular Network Infrastructure. <https://osmocom.org/projects/cellular-infrastructure/wiki>
- [44] P1 Security. 2020. P1 Telecom Fuzzer. <https://www.p1sec.com/corp/products/p1-telecom-fuzzer-ptf/>
- [45] Shinjo Park. 2017. SCAT: Signaling Collection and Analysis Tool. <https://github.com/fgsect/scat>
- [46] Shinjo Park, Altaf Shaik, Ravishankar Borgaonkar, and Jean-Pierre Seifert. 2016. White Rabbit in Mobile: Effect of Unsecured Clock Source in Smartphones. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 13–21.
- [47] David Rupperecht, Adrian Dabrowski, Thorsten Holz, Edgar Weippl, and Christina Pöpper. 2017. On Security Research Towards Future Mobile Network Generations. (oct 2017). arXiv:1710.08932 <http://arxiv.org/abs/1710.08932>
- [48] David Rupperecht, Kai Jansen, and Christina Pöpper. 2016. Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*.
- [49] David Rupperecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. 2019. Breaking LTE on Layer Two. In *2019 IEEE Symposium on Security and Privacy (SP)*.
- [50] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2020. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, 2020*. <https://doi.org/10.14722/ndss.2020.23096>
- [51] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182.
- [52] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. 2015. Practical attacks against privacy and availability in 4G/LTE mobile communication systems. (2015). <http://arxiv.org/abs/1510.07563>
- [53] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. 2019. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24–27, 2019*. The Internet Society. <https://www.ndss-symposium.org/ndss-paper/periscope-an-effective-probing-and-fuzzing-framework-for-the-hardware-os-boundary/>
- [54] Daehyun Strobel. 2007. IMSI Catcher. *Chair for Communication Security, Ruhr-Universität Bochum* (2007).
- [55] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. 2018. ATtention Spanned: Comprehensive Vulnerability Analysis of {AT} Commands Within the Android Ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*. 273–290.
- [56] Nathan Voss. 2017. afl-unicorn: Fuzzing Arbitrary Binary Code. <https://hackernoon.com/afl-unicorn-fuzzing-arbitrary-binary-code-563ca28936bf>
- [57] Ralf-Philipp Weinmann. 2012. Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks. *USENIX Workshop on Offensive Technologies* (2012).
- [58] WikiChip. 2020. Helio X10 (MT6795) - MediaTek. <https://en.wikichip.org/wiki/mediatek/helio/mt6795>
- [59] Ben Wojtowicz. [n.d.]. OpenLTE. <http://openlte.sourceforge.net/>
- [60] Michael Zalewski. 2016. Technical "whitepaper" for AFL-fuzz. <http://lcamtuf.coredump.cx/afl/>
- [61] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313. <https://doi.org/10.1109/ACCESS.2018.2851237>