BaseBridge: Bridging the Gap between Over-The-Air and Emulation Testing for Cellular Baseband Firmware

Daniel Klischies*, Dyon Goos†, David Hirsch*, Alyssa Milburn‡, Marius Muench§ and Veelasha Moonsamy*
*Ruhr University Bochum, †Vrije Universiteit Amsterdam, ‡Independent, §University of Birmingham
firstname.lastname@rub.de, d.w.goos@vu.nl, amilburn@zall.org, m.muench@bham.ac.uk

***Abstract*—Current approaches for emulating cellular basebands inherently fall short in comparison to over-the-air testing due to their limited support for the complex peripherals involved in a modern baseband, such as DSPs, SIM cards and RF frontends. Improving such support is a daunting task, requiring deep reverse-engineering which is extremely time consuming – resulting in slow progress. Consequently, techniques such as fuzzing are only able to find relatively shallow bugs, since they are unable to reach the states required for the majority of the baseband to function.**

To fill this gap, we propose BASEBRIDGE, which enables far more comprehensive simulation of baseband behavior by *restoring* relevant state from memory dumps of real devices. Our prototype implementation supports baseband firmware from two major vendors (MediaTek and Samsung), and – in contrast to current state-of-the-art emulators – correctly responds to 97% of tested RRC and NAS messages while improving coverage by an average factor of 2.41 (Samsung) and 5.54 (MediaTek). BASEBRIDGE also passes several LTE conformance tests. Our empirical evaluation demonstrates that this enhanced fidelity enables faster discovery of a wider range of bugs thanks to the scalability of emulation; our fuzzing campaign shows that coverage improves by a factor of 2.3–5x overall, and by a factor of 9.0–22.5x for functionality targeted by our approach. BASEBRIDGE unveiled 5 new vulnerabilities, which we have disclosed to affected vendors.

1. Introduction

Basebands, dedicated processors in smartphones' chipsets, are essential components that establish connectivity to cellular networks. The firmware running on baseband processors has become an increasingly lucrative target for threat actors [38], which in turn, has led to more active research on the identification of vulnerabilities in baseband firmware in recent years. Researchers have proposed fuzzers [13], [24], approaches to validate the correct implementation of cellular network protocol specifications [7], [23], and systematic evaluation of basebands' handling of various protocol violations [25], [34]. Most of these approaches work "Over-the-air (OTA)", that is, they test physical User Equipment (UE) by supplying the test signals using a small-scale cellular network. This requires a test base station, usually using a Software Defined Radio (SDR) and open source implementations of a cellular base station and core network. This has two advantages: First, OTA testing offers a high *accuracy*, as vulnerabilities found in the lab environment will apply in real-world scenarios as well – given that the environment is a scaled down, but otherwise mostly identical replication of the real world. Second, OTA testing enables the assessment of a *broad* range of complex functionalities supported by the baseband.

However, OTA testing also has several inherent downsides. To scale OTA testing horizontally, additional UEs and SDRs must be acquired, posing a significant financial challenge. Furthermore, OTA testing setups offer limited introspection into the behavior of the baseband firmware. Without the tools available only to the baseband and UE manufacturers, external researchers often cannot distinguish between, e. g., whether a crash is caused by a denial of service or remote code execution vulnerability. Lastly, OTA testing is relatively slow. Recent state of the art approaches require 2 to 10 seconds to test a baseband's behavior on a single packet [34], [40]. In particular, experiments that rely on testing large number of packets – such as fuzzing – are severely hampered by this limitation of OTA testing.

Several recent works address the aforementioned limitations of OTA testing by executing the baseband firmware in an emulator, rather than on a physical device [17], [32]. These emulators can be run on commodity Linux servers, enabling horizontal scaling by launching multiple emulators in parallel. They further improve performance by a factor of 100 to 10000 compared to OTA testing, as they can take and restore snapshots of the emulated baseband's state. Lastly, emulators enable dynamic analysis, allowing researchers to place arbitrary breakpoints and introspect the baseband's memory and registers at any point during execution.

However, existing emulation approaches fall short in scenarios where OTA approaches excel. In particular, existing emulation approaches are typically only able to comprehensively simulate the baseband's behavior during parsing of incoming packets, and frequently stop execution shortly after. This limitation is illustrated by the fact that none of the existing emulation approaches are able to generate a response to any Downlink (DL) packet on two essential protocol layers of the 4G specification, Radio Resource Control (RRC) and Non-Access Stratum (NAS). Current emulation approaches therefore enable limited, but not broad, security evaluation of baseband firmware. We theorize that this is

due to the enormous amount of state that a baseband must take into account when processing incoming packets.

In general, basebands can only process a packet if the baseband is connected to a cellular network. Depending on the packet, processing may require that the baseband and the cellular network have previously successfully negotiated an encryption algorithm for the connection or that the cellular network is located in a certain country. Likewise, the baseband may require state that is derived from peripheral devices, such as the Subscriber Identity Module (SIM) card's International Mobile Subscriber Identity (IMSI), or signal strength measurements conducted by the Radio Frequency (RF) frontend. As existing emulators neither emulate the cellular network, nor complex peripherals such as SIM cards or RF frontends, the emulated firmware lacks all of this state. Instead, it behaves like it is not connected to a network, does not contain a SIM card, and does not have the required peripherals to perform any RF signal measurements. Building this state inside an emulator would require emulating these complex peripherals, and simulating the entire behavior of a cellular network, down to the RF signal level. As commercial basebands are entirely closed source, this not only requires huge reverse-engineering efforts, but also must be implemented separately for each baseband manufacturer, and potentially even for each baseband model.

To close the gap between OTA and emulated baseband security testing, we propose BASEBRIDGE, to enable *broad* and *accurate* emulation with faster, scalable introspection capabilities. BASEBRIDGE is generic and therefore portable between various baseband manufacturers and models, and requires no manual reverse-engineering to implement the behavior of additional complex peripherals. The novelty of BASEBRIDGE lies in its ability to transfer state of a physical device that is already connected to a cellular network, and continue the emulation from this state. We show that our implementation of BASEBRIDGE can process 97% of tested RRC and NAS messages in a way that matches the response of a physical baseband in an OTA scenario. Furthermore, we demonstrate that BASEBRIDGE is the first baseband emulation approach that successfully passes several cellular protocol conformance tests that are normally used to qualify physical basebands for mass production, further illustrating its accuracy in replicating the behavior of a physical baseband. We further evaluate BASEBRIDGE by fuzzing baseband firmware of two major baseband manufacturers, i.e., MediaTek and Samsung, improving coverage by as much as 9.0–22.5x in targeted cases. These improvements uncover 5 new and 3 previously known vulnerabilities, including several remote code execution vulnerabilities, enabling an attacker to take over full control of affected basebands over-the-air.

In summary, we make the following contributions:

- We propose BASEBRIDGE, an approach to transfer state from physical devices, allowing researchers to leverage the advantages of emulation-based baseband firmware testing, while exploring functionality that requires state only available in OTA testing.
- We demonstrate BASEBRIDGE's real-world applicability by implementing a prototype, supporting basebands from two major vendors, based on an existing state-of-the-art baseband emulator.
- We evaluate BASEBRIDGE both qualitatively, demonstrating that it reproduces the behavior of physical basebands accurately, and quantitatively via fuzzing, uncovering a total of 8 vulnerabilities.

The source code to BASEBRIDGE along with supporting artifacts will be made publicly available at https://github.com/FirmWire/BaseBridge.

Coordinated Disclosure: All vulnerabilities presented in this paper have been disclosed to, and confirmed by, affected manufacturers. Patches are currently being developed and deployed to end-users.

2. Background

2.1. Cellular Protocol Stacks

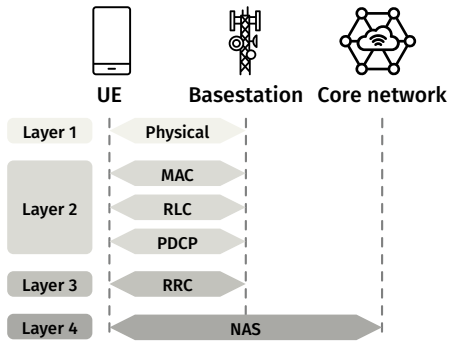

Figure 1. Overview of the LTE DCCH protocol stack.

Cellular networks consist of both, a network side, i.e., the Base Tranceiver Station (BTS), sometimes also referred to as BTS or Evolved Node B (ENB), connected to a core network, as well as consumer devices, so called UEs. Most UEs contain a baseband for handling the cellular communication. To maintain interoperability all over the world and between different vendors, cellular networks rely on common protocols. The cellular protocols are composed of several layers that together make up the celullar protocol stack. Various cellular protocol stacks are specified, ranging from the legacy Global System for Mobile (GSM) to the latest 5G standards. At the time of writing, Long Term Evolution (LTE) is the most prevalent of these [16].

LTE is divided into multiple channels, depending on whether the packets carried by the channel are logically unicasted or broadcasted. The LTE unicast channel, implementing most of the security relevant functionality, is the Dedicated Control Channel (DCCH) channel. The LTE DCCH protocol stack consists of NAS and RRC, several Layer 2 protocols (Packet Data Convergence Protocol (PDCP), Radio Link Control (RLC), Medium Access Control (MAC)) and a Physical Layer, as shown in Figure 1.

Unlike protocols from stacks such as TCP/IP, the individual layers rely on the presence of both lower and upper layers and are highly interconnected: e. g., every protocol below RRC is configured by RRC, and yet NAS messages are transported via RRC.

2.2. Modem firmware

The firmware for baseband modems is built using a Real-Time Operating System (RTOS). Such firmware [17], [23], [32] typically consists of the core RTOS, drivers, and cellular specific code. The majority of functionality is implemented as a large number of *tasks*. Some tasks implement OS functionality, while others contain cellular-specific functions, and communicate via message passing. The latter often implement specific pieces of the different layers within the cellular stack, and inherently work closely with many other tasks. For example, a Layer 3 RRC task may communicate with both Layer 2 and Layer 1 tasks. However, these tasks are not isolated from each other, neither technically nor even by design; protocol state is expected to be consistent between tasks, and actual state is often shared between them. Firmware state can also be stored in many forms and locations, including:

Stacks: Stacks for each task, as well as potential stacks used by the RTOS kernel.

Global variables: Used for storing data belonging to individual tasks and global data for the kernel, such as information about the currently running task.

Heaps & Arenas: Dynamically allocated by tasks/kernel.

Registers: Baseband processor registers, e.g. the program counter, stack pointer and general purpose registers.

Peripherals: Configuration and state of peripherals connected to the baseband, which typically use MMIO, interrupts, and DMA to interact with the baseband.

Reverse-engineering even small portions of such firmware is challenging due to the sheer size of the firmware, the highly complex and interconnected tasks, the large amount of state of different types, and the wide range of devices with different firmware versions which are built and configured in different ways.

3. Motivation

We motivate our work with an explanatory case study to provide insights into the implementation of a Commercial-off-the-shelf (COTS) baseband and derive concrete challenges which our solution must overcome. Specifically, we discuss how a MediaTek baseband firmware responds to a Counter Check message and how state affects its processing.

RRC Counter Check mechanism. The purpose of a RRC Counter Check message is to verify the amount of data transmitted on each data channel. Although the Counter Check itself is a RRC message, the respective counter is kept by the PDCP layer, already highlighting the interconnection between different layers in the cellular protocol stack.

```
1  rrc_conn_state_struct* conn_state = 0x63b3d3f4;
2  state_machine_struct[] state_machines = 0x629e4474;
3
4  void errc_conn_main(void *msg_ptr) {
5    /* ... */
6    int i = 0, state_id = conn_state->current_state;
7
8    for (i; state_machines[i]->msg_id != -1; i++) {
9      if (state_machines[i]->msg_id != msg_id) continue;
10     int current_proc = state_machines[i][state_id];
11     do {
12       switch(current_proc) {
13         case 1:
14           errc_conn_est_main(msg_ptr, &state_id,
15             &current_proc);
16           break;
17         case 2:
18           errc_conn_smc_main(msg_ptr,&state_id,
19             &current_proc);
20           break;
21         /* ... */
22         case 11:
23           errc_conn_lwa_main(msg_ptr, &state_id,
24             &current_proc);
25           break;
26         default:
27           trap(13);
28       }
29       /* ... */
30       conn_state->current_state = state_id;
31     } while (current_proc != 0);
32     return;
33   }
34 }
```

Listing 1. Example of internal state usage (simplified and adapted from `errc_conn_main`).

To process a RRC counter check message, the UE requires at least an initial connection (i.e., it is in the `RRC_CONNECTED` state). Moreover, ciphering and integrity protection of the RRC connection must have been activated. Although the security is configured using the RRC's Security Mode Command, ciphering and integrity protection itself is handled within the PDCP layer [4]. The ability of an (emulated) UE to correctly process a Counter Check message depends on all these different layers being in a consistent and appropriate state, which current emulation-based approaches do not support.

Parsing the Counter Check in the baseband. Upon receiving an incoming Counter Check message, it is forwarded from the PDCP task to the RRC task[1]. The message is decoded and a new internal message, depending on the type of the received message, is passed to the RRC task. The RRC task performs several checks; for example, it checks whether the UE is in a `RRC_CONNECTED` state, as described before. If any check fails, the message is discarded.

In the MediaTek baseband firmware we chose for this example, further processing of the message is controlled by a function implementing a finite state machine, as shown in Listing 1. This relies on two global variables: a struct which stores a variety of state data (Line 1), and an array of structs (Line 2) containing the state machine's definition. The state machine needs to handle incoming messages differently

1. MediaTek uses `e` (presumably `evolved`) as a prefix for the LTE tasks and functions, i. e., the RRC task is called ERRC.

depending on the current state of the baseband. To correctly handle a RRC Counter Check message, the baseband will need to reach the `errc_conn_smc_main` function (L18), which only occurs if `current_proc` is equal to 2. In other states, the message would not be processed further, but would be silently discarded or even rejected.

To know which combination of `rrc_conn_state` and `state_machine_struct` would allow for reaching this procedure during execution, an analyst would have to overcome several challenges, which we elaborate on below.

Size of baseband firmware. Through static analysis, we find that the state data structure (Line 1) has a (minimum) length of 1546 bytes and 1153 references. The high number of references shows that this memory is shared between several functions. Although the high-level state is only a single field of this structure, certain conditions require other fields to be checked or updated[2]. If the relevant path through the firmware is not known beforehand, the high number of similar looking code paths – some dependent on the state – make manual reverse-engineering infeasible at scale. This illustrates our first challenge: the sheer amount of data structures and code blocks in baseband firmware.

In our example, the outer loop (L8–L9) retrieves the state machine definition corresponding to the current message. The definition contains fields for every possible state, including which function should be executed next (L10). The inner loop executes the selected function, which is repeated until the end of execution is signaled (L11–L31). The callees can change the value of `state_id` to influence the processing of further messages, as well as the value of `current_proc` which causes a different function to be called in the next iteration of the inner loop (L14, L18, L23). Again, if the state or message does not match the state machine definitions, the message is either discarded or a trap is executed (L27, L32).

Interdependency and complexity of baseband firmware. Furthermore, complex interdependencies between variables make it infeasible to restore the state by setting a few variables to manually reverse-engineered constant values. Moreover, the correct values are usually not visible via static analysis and without BASEBRIDGE, no means for dynamic analysis exist. Other solutions require manual modeling of the required state which itself mandates a deep understanding of the firmware. Finally, recovering data structures is an unresolved challenge in binary analysis and we have shown that understanding the baseband firmware's data structure is essential for its functionality. This high level of interdependency and complexity marks our second challenge.

In our example, the callee is responsible for the actual handling of the message. It first checks whether security is activated for the connection. If not, the message is discarded, as described in Section 2.1. Execution is then passed to the PDCP task to query the counter values. This might require the PDCP task to pass execution to a peripheral, a coprocessor dedicated to Layer 2 tasks. Finally, execution is passed back to the RRC task and the Counter Check Response message is created, encoded, and passed on to the PDCP task for transmission.

Non-Scalability of Manual Analysis. Lastly, while we have manually reverse-engineered this particular example for the MediaTek firmware, other vendors and parts of the stack could (and would) implement similar functionality differently. This brings us to our third challenge: These efforts must be repeated for every vendor or even multiple times for the same vendor, e. g., for distinct baseband models, highlighting scalability shortcomings in existing solutions.

4. BASEBRIDGE

The novelty of our proposed approach, BASEBRIDGE, is that we can reach a desired state on a physical baseband during normal operation, and then *transfer* that state into an emulator which is capable of running the baseband's firmware – even when the emulator cannot reach the aforementioned baseband's state, possibly due to missing peripherals.

BASEBRIDGE is comprised of two concrete high-level components, as shown in Figure 2. (1) First, it obtains the contents of the baseband processor's memory from a real phone after the baseband has already reached the desired state. (2) Second, as the memory dumps cannot be restored directly, BASEBRIDGE implements a *Dynamic Memory Restoring* algorithm to identify and restore the necessary portions of memory into the emulated memory state. Ultimately, this allows to observe the baseband's behavior in the presence of state and provides a faithful approximation of the physical baseband used to extract the memory dump.

4.1. Extracting State from Physical Devices

To enable stateful emulation of the baseband, we need to obtain the firmware and the state of the baseband from a COTS UE. To fully transfer the complete state listed in Subsection 2.2, all of these aspects would have to be extracted from a physical UE and restored into the emulator. This is inherently challenging; for example, extracting peripheral state (such as storage devices or the RF frontend) without manual per-peripheral effort is likely to be impractical.

However, a key insight is that it is neither required nor necessarily desirable for BASEBRIDGE to obtain all of the low-level state; we only need to restore portions which are (or might be) relevant to the connection state. Global variables and heap memory are used by tasks to store information such as what network they are connected to, the current phase of connection establishment or whether authentication is enabled (the *connection state*), and so these regions may be relevant. In contrast, the contents of the stack and the program registers only reflect the progress in processing the current message, and hence do not contain any persistent state. Finally, we note that global variables used by the kernel itself are used to store information on the state of the RTOS, and not the connection state.

2. For the sake of readability, this is not shown in Listing 1.

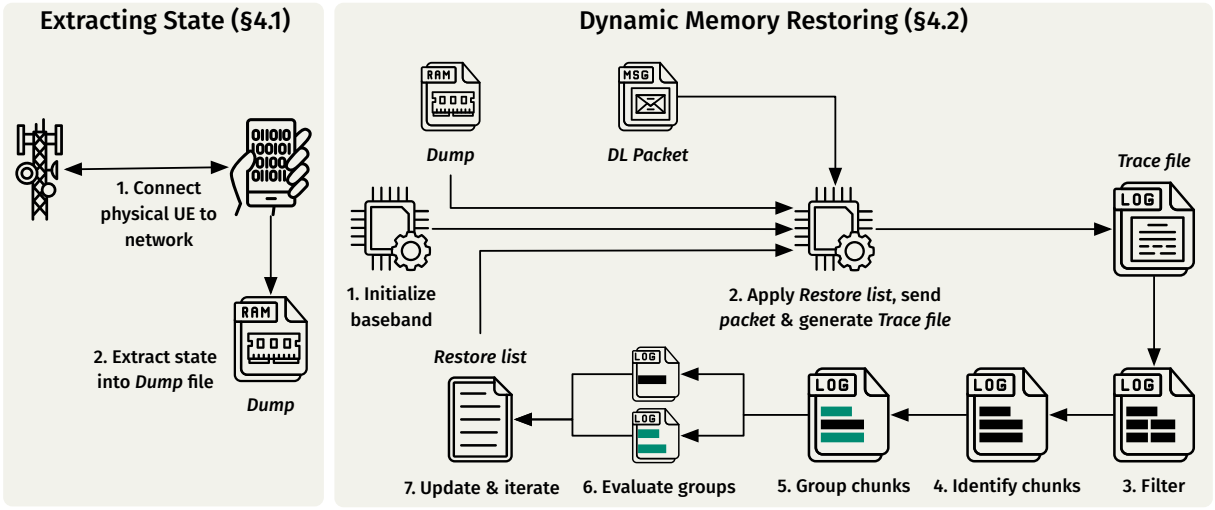

Figure 2. Overview of BASEBRIDGE.

As such, BASEBRIDGE can rely on the emulated firmware's state after boot for much of the baseband state, including kernel global variables, stack memory, and peripheral state. We require only a memory dump containing global and heap memory contents. We propose three concrete approaches which can be used by BASEBRIDGE to obtain such a dump from a physical baseband:

(i) Manufacturer tooling. Memory can be read during execution using debugging tools specific to the baseband's manufacturer. However, access to such tools is typically exclusive to smartphone manufacturers and protected by Non-Disclosure Agreements (NDAs). While it may be possible to reverse-engineer the relevant debug APIs, such open-source tooling currently only exists for Qualcomm basebands, and it is unable to dump processor register contents [33].

(ii) Leveraging vulnerabilties to inject a debugger. A debugger can be injected by exploiting (pre-existing) issues, for instance OTA Remote Code Execution (RCE) vulnerabilities or flaws in the baseband firmware update process. This approach has two disadvantages: First, it requires a vulnerability. We expect baseband manufacturers to fix such vulnerabilities quickly following coordinated disclosure, preventing analysis of future firmware versions. Second, it is likely to be difficult to preserve baseband stack and register contents, since the debugger runs on the baseband itself and thus affects its run-time state.

(iii) Crash dumps. A method that is widely-supported across different baseband models and manufacturers is leveraging the baseband's built-in crash dump functionality. This dumps memory contents to non-volatile memory during baseband crashes, caused by e. g., invalid memory accesses or violated assertions. Some smartphone manufacturers, like Samsung, even provide debug menu options to trigger the generation of such dumps. However, their generation requires raising an interrupt, overwriting stack contents, and resulting dumps typically do not contain register contents.

Observation: Several practical methods allow to extract main memory contents from a commercial baseband, all of which include the global variables and heap memory, thus providing sufficient information about the connection state. Register contents, peripheral state, and precise stack dumps, on the other hand, are unrealistic to obtain in most scenarios. Thus, BASEBRIDGE must be able to work with the resulting *imprecise* memory dumps.

4.2. Dynamic Memory Restoring

Due to imprecise memory dumps, related to the observations made in the previous paragraph, BASEBRIDGE cannot load the memory dump as a whole into the emulator, as doing so would cause the emulator to crash early before processing any packets. Therefore, BASEBRIDGE should (ideally) only restore portions of the memory which are relevant to the connection state inside the emulator. To do this, we need a tractable approach for identifying memory regions relevant to the connection state, that can scale up to the complexity and interdependency of baseband firmware.

Unfortunately, identifying relevant memory using static analysis alone would require a prohibitive amount of manual effort and is likely to yield a high number of false positives, as global variables and dynamically allocated structures are commonly interleaved with other data (e.g., RTOS state).

Therefore, BASEBRIDGE instead identifies the memory regions that contain the relevant connection state *dynamically* and attempts to restore the connection state in the emulator *iteratively* from the imprecise memory dump.

At a high level, we inject OTA packets into the emulator and log which regions (*'chunks'*) of memory are accessed during execution alongside metadata for each access. We use the access data to create a *restore list* to selectively restore relevant memory from an imprecise memory dump. To create a restore list for a given baseband firmware, we develop a general algorithm for *Dynamic Memory Restoring* that can be combined with vendor specific heuristics.

This allows BASEBRIDGE to efficiently handle the subtle differences in the baseband RTOS design.

Our approach starts with an empty restore list and iteratively determines which chunks are relevant for the connection state. For this, our *Dynamic Memory Restoring* algorithm executes the following steps:

(i) Core initialization. We begin by executing the firmware under test in an emulator up to the point where it would be ready to receive incoming OTA packets from a basestation. This provides basic initialization for the baseband's RTOS and its tasks, but leaves all connection state uninitialized.

To speed up *Dynamic Memory Restoring*, we take a snapshot of the emulated baseband at this time, which will serve as a basis for all other steps.

> ***Running example:*** Recall the example from Listing 1. After core initialization, the RRC task would be ready to process incoming messages, but all state variables are likely to be uninitialized or in a disconnected state, reflecting a fresh boot of the baseband.

(ii) Trace generation. We restore the snapshot and select a DL packet for injection into the baseband. Immediately before the first injected packet is processed, we restore all the chunks in the restore list from a physical baseband's memory dump; note that this list is initially empty.

Then, we begin execution and log the addresses of all memory accesses (reads and writes) in a *trace file*, along with the program counter and the return address at the time of access. We stop the emulation either if the emulated firmware encounters a crash (e.g., due to accessing unmapped memory or hitting an assertion) or finishes processing the injected packet and returns to the main loop. During execution, we also monitor for Uplink (UL) *response* packets, which would be sent from the UE to the basestation.

> ***Running example:*** We inject a RRC packet into the baseband which triggers an internal Counter Check message. BASEBRIDGE logs the accesses to the current_state field of conn_state on line 6 and to the various offsets of state_machines on line 10.

(iii) Filtering. Where applicable, vendor-specific heuristics can now be applied to create a filtered trace file and speed up *Dynamic Memory Restoring*. The input to the filtering process is the trace file, a list of memory regions or chunks to be excluded, and (optionally) the memory dump from the physical phone. Example heuristics include:

- Exclusion of task code and stack regions.
- Exclusion of regions known ahead of time to contain only kernel memory or other irrelevant data.
- Exclusion of memory which is overwritten without being read, as this cannot be relevant state.
- Exclusion of regions which are not included in the memory dump as these cannot be restored (e.g., regions corresponding to peripherals).

> ***Running example:*** We know ahead of time that the code of Listing 1 is static and will never change, regardless the processed messages or connection state. Thus, we filter the memory regions holding the code from the trace file.

(iv) Identification of memory chunks. Where sufficient heuristics are either unavailable or insufficient, BASEBRIDGE continues its analysis by traversing the filtered trace file to identify continuous memory chunks. A *memory chunk* is a range of consecutive memory addresses that have been accessed in the same way (i.e., read or write access), during the current emulation run. Intuitively, in many cases a memory chunk will correspond to a structure or an array.

We expect that structures and arrays will never store connection state and kernel state at the same time. Thus, in Step 2, chunks will either be entirely restored or not at all.

> ***Running example:*** If during execution another function accesses a field that is adjacent to the current_state field of the conn_state structure, the addresses of both fields will be merged into a single memory chunk.

(v) Grouping of memory chunks. At this stage, the simplest approach to identify connection state would be to re-run the emulator with every possible combination of restored memory chunks. For every combination, BASEBRIDGE could check whether the emulation produces the expected response. However, typical trace files contain several thousand accessed chunks and the resulting combinatorial explosion makes this approach impractical. Instead, we reduce the number of test cases by identifying groups of chunks that likely need to be restored together, by determining which chunks are accessed within the same functions.

For each memory chunk, we use the filtered trace file generated in the previous step to determine the list of functions which either (a) directly access that chunk or (b) call a function accessing that chunk. We then create a group containing both the memory chunk as well as all other memory chunks that are (directly) accessed by those functions. This results in a set of groups which approximate the dependencies between the memory chunks as expressed by the functions using them.

> ***Running example:*** We see that errc_conn_main accesses both the conn_state->current_state variable and the state_machines array. Thus, we add the memory chunks for both conn_state->current_state and state_machines to the same group.

(vi) Per-group evaluation. Now, BASEBRIDGE has several candidate groups available for restoring and each group consists of multiple memory chunks that are semantically connected with each other. However, we still need to determine whether a group represents memory chunks containing *connection state*, and thus should be added to our restore list.

For each group of memory chunks, we restart the emulator from the snapshot with the memory chunks of that group added to the restore list (which is empty during the first iteration of *Dynamic Memory Restoring*). We measure how many unique basic blocks are accessed and store all log messages written by the emulated baseband. After the packet has been processed (or emulation aborted), we restart the emulator, reset the restore list and repeat this process for the next group, continuing until all groups have been tested.

To assess which groups of memory chunks are the most promising to restore, we use feedback collected from the emulator as a measure of success; groups with relevant state will enable the emulator to proceed further with processing packets. To identify the most promising groups, we use the following metrics: (a) execution of new basic blocks, (b) addition of new log messages, and (c), prevention of crashes.

Whenever a group fulfills at least one of these criteria, and has not lead to a significant decrease of coverage, i. e., has not disabled previously working functionality, we add the group to our baseline restore list so that this memory will always be restored in future executions of the emulator.

> ***Running example:*** Restoring state_machines and conn_state−>current_state will likely allow emulation to proceed into one of the functions inside the switch-case statement, executing new basic blocks and potentially avoiding the trap in the default case (l. 27).

(vii) Iteration. After per-group evaluation, we perform the next iteration starting from Step 2 with the new restore list. Steps 2–7 are repeated until we reach a fixed point where no new memory accesses are found and *Dynamic Memory Restoring* has produced a final restore list.

5. Implementation

To demonstrate the feasibility of our approach, we implement a prototype of BASEBRIDGE on top of the FIRMWIRE platform. It provides support for full-system emulation of Samsung and MediaTek basebands [17] and we likewise implement BASEBRIDGE for baseband firmware of both vendors. Our BASEBRIDGE prototype consists of approximately 2250 lines of Python including the core algorithm, orchestration features, and additions to FIRMWIRE's vendor plugins. In the following, we detail noteworthy implementation details and performance optimizations.

5.1. Implementation Details

Creating and parsing crash dumps. For both MediaTek and Samsung, we rely on manually creating crash dumps via the according vendor-specific debug menu. However, these crash dumps use proprietary formats, so additional preprocessing is required for mapping memory contents of a crash dump to the memory layout of the emulated baseband.

For MediaTek, we manually reverse-engineered the crash dump format to establish this mapping. We take a simpler approach for Samsung crash dumps: we search the crash dump file for known patterns from the emulation state, and use these to determine the correct mapping.

Emulating downlink and uplink messages. BASEBRIDGE extends FIRMWIRE to include support for supplying DL messages and observing UL messages using the GSMTAP format, which enables both dynamic communication as well as use of the well-established `pcap` format. This is used to provide appropriate packets during the initial trace generation, and can also be used during later dynamic analysis. This allows for two-way communication between the emulated UE and a mocked up base station, and allows us to observe the network traffic via WIRESHARK.

Generating trace files. FIRMWIRE's core is based on Platform for Architecture-Neutral Dynamic Analysis (PANDA) [9]. As such, the framework exposes a rich API for run-time hooks and on-the-fly coverage generation. We leverage PANDA to install memory hooks to track memory read and write accesses. This allows us to create trace files and enrich the memory access information with metadata such as current program counter value or return address.

5.2. Performance Optimizations

Parallelization. Various steps of *Dynamic Memory Restoring* can benefit from parallelization, as the individual steps do not necessarily need to execute their tasks sequentially. In particular, per-group evaluation (Step 6) requires executing the emulator a significant amount of times under different configurations of restored groups. We parallelized this step and allow a user-defined amount of emulation instances to run at the same time to speed up this process.

TABLE 1. IMPLEMENTED HEURISTICS BY VENDOR.

ID	Excluded memory objects	MediaTek	Samsung
H1	Memory outside of memory dump	✓	✓
H2	RTOS & tasks' code	✓	✓
H3	Tasks' stacks		✓
H4	Irrelevant heap data		✓
H5	Memory written to before first read		✓
H6	Global message queue data		✓
H7	Global event data		✓
H8	Global timer data		✓

Vendor-Specific Heuristics. A core optimization of BASEBRIDGE is the option to apply heuristics to exclude memory chunks from the restore candidates. We list the heuristics we implemented and the vendors we applied them to in Table 1.

We utilize a wider variety of heuristics for Samsung baseband firmware due to the structure of the underlying RTOS. In ShannonOS (Samsung's proprietary RTOS) some task state is typically well separated from connection state, which allows the creation of simple yet effective and scalable heuristics through FIRMWIRE's `PatternDB` system.

We extended BASEBRIDGE's Samsung vendor plugin with a scalable heap parser capable of identifying the heap

metadata and the allocated heap regions from the crash dump. With this knowledge, we expand the *Dynamic Memory Restoring* algorithm to ensure that new heap allocations are placed in a new region of memory, avoiding conflicts.

6. Evaluation

To evaluate BASEBRIDGE, we first want to analyze improvements in terms of emulation accuracy. For this, we replay DL messages and compare the UL responses between BASEBRIDGE, plain FIRMWIRE as well as a physical UE (Subsection 6.1). To understand the gains from improved accuracy to fuzz testing, we evaluate BASEBRIDGE's performance during multiple fuzzing campaigns (Subsection 6.2). Finally, to demonstrate that BASEBRIDGE enables scalable and stateful security testing of emulated UEs, we showcase and discuss discovered vulnerabilities (Subsection 6.3).

Experimental setup. We focus our experiments on parts of the LTE implementations. For this, we extended the existing MediaTek RRC FIRMWIRE modkit task and created a FIRMWIRE modkit task for Samsung from scratch. These tasks allow to both inject RRC and NAS messages sent over the DCCH, Broadcast Control Channel (BCCH), Common Control Channel (CCCH) and Multicast Control Channel (MCCH) LTE channels. In contrast to plain FIRMWIRE, we did not need to manually reverse-engineer and hardcode state-related addresses, greatly reducing the required effort.

As targets for our evaluation, we select sample firmwares for the MediaTek MT6768 baseband and Samsung Exynos 9820 basebands[3]. We create a crash dump from a physical Samsung Galaxy A41 and S10e connected to srsRAN via LTE, and use them as the memory dumps for our *Dynamic Memory Restoring* algorithm.

6.1. Qualitative Evaluation

To evaluate BASEBRIDGE, we compare UL responses to various DL packets sent over the DCCH channel. Within Subsection 6.1.1, we evaluate our *test packets*: We built a broad set of test packets, containing representative RRC and NAS packets on the DL DCCH channel. Each packet contains default values and the minimum required Information Elements (IEs). We send each packet to BASEBRIDGE, plain FIRMWIRE, and the physical UE to compare the UL packets sent in response. Naturally, we consider responses from the physical UE as our ground truth.

Within Subsection 6.1.2, we verify BASEBRIDGE and FIRMWIRE against several *conformance tests*, specified by the 3GPP and originally intended to verify the conformance of COTS UEs to the cellular specifications [2], [3]. Each conformance test consists of one or multiple RRC and NAS DL messages that we send to both emulators, observing their UL responses and verifying them against the criteria defined by the test. Compared to our own test cases, the conformance tests verify fewer different network packet types, but do so in a more in-depth way by triggering specific functionalities via clearly defined packets, sometimes even involving multiple packets send and received in sequence.

3. FW versions `A415FXXS2DWA2` and `G970FXXSGHWC2`

6.1.1. Test Messages. The results from replaying our DL *test packets* are shown in Table 2. This table only shows DL messages that trigger a UL response from the physical UEs (i.e., they are neither silently processed nor discarded). Appendix A lists all packets. To estimate the effectiveness of BASEBRIDGE, we additionally compare the number of basic blocks executed between FIRMWIRE and BASEBRIDGE during the processing of our *test packets*. The relative difference in discovered basic blocks between FIRMWIRE and BASEBRIDGE is shown in column δ_{BB}.

We find that, with the exception of the emulated MediaTek baseband not responding to the RRC ReconfigurationRequest, both emulated basebands respond with the same UL packets as the physical UE. This is in stark contrast to FIRMWIRE, where the emulated basebands do not respond to any DL packet. Upon inspecting the log traces, we find that Samsung firmware validates the RRC state immediately after parsing the DL packet. As FIRMWIRE is missing this state, the baseband will discard the DL message in almost all cases immediately without any message processing. The MediaTek firmware has a similar check, which is passed for some messages as the original FIRMWIRE RRC implementation manually populates select state variables. However, it discards all packets shortly after they have been parsed due to further (message-specific) state checks.

Upon further analysis, we notice that BASEBRIDGE's emulated MediaTek baseband does not respond to the RRC ReconfigurationRequest due to failure of a component named `L2COPRO`. We suspect that this stands for Layer-2 co-processor, a dedicated component used in baseband architectures to accelerate integrity protection and compression of packets. As this peripheral is missing in the emulator, the request fails and the emulated baseband firmware crashes. We note that such dependencies on specific peripherals are an exception – all other DL packets are processed and responded to. This example also showcases the architectural differences between baseband vendors, as the Samsung baseband correctly responds to the RRC Reconfiguration-Message when emulated via BASEBRIDGE.

Overall, our evaluation shows that BASEBRIDGE increases the number of covered basic blocks by an average factor of 2.41x or 5.54x (depending on baseband vendor). This illustrates the huge impact of automatically restoring connection state. In the context of security testing, this added coverage corresponds to additional attack surface that is now reachable for evaluation and fuzz testing.

6.1.2. Conformance Tests. Our *test packets* provide a broad evaluation but do not necessarily trigger semantically complex functionality. Thus, we also evaluated BASEBRIDGE using *conformance tests*. The 3GPP provides extensive documentation on conformance tests, meant to qualify COTS UEs before production. For LTE, TS36.523-1 and TS36.508 specify these tests [2], [3]. Each conformance test specifies

TABLE 2. UL RESPONSES TO DL MESSAGES BETWEEN THE UE, FIRMWIRE AND BASEBRIDGE. A UL RESPONSE IN GREEN DEPICTS A MATCHING RESPONSE BETWEEN THE UE AND BASEBRIDGE, WHEREAS AN UL RESPONSE DEPICTED WITH A "-" CORRESPONDS TO A MISSING UL REPONSE.

Message	Vendor	Expected Response	Reponse FIRMWIRE	Response BASEBRIDGE	BB_{cov} FIRMWIRE	BB_{cov} BASEBRIDGE	δ_{BB}
RRCConnectionReconfiguration (RRC)	Samsung	RRCConnectionReconfigurationComplete	-	RRCConnectionReconfigurationComplete	3859	6767	1.75
	MediaTek			-	220	1757	7.99
SecurityModeCommand (RRC)	Samsung	SecurityModeFailure	-	SecurityModeFailure	2210	3130	1.42
	MediaTek				203	468	2.31
UECapabilityEnquiry (RRC)	Samsung	UECapabilityInformation	-	UECapabilityInformation	2226	3885	1.75
	MediaTek				214	2048	9.57
CounterCheck (RRC)	Samsung	CounterCheckResponse	-	CounterCheckResponse	2235	3870	1.73
	MediaTek				202	556	2.75
UEInformationRequest-r9 (RRC)	Samsung	UEInformationResponse-r9	-	UEInformationResponse-r9	2325	2990	1.29
	MediaTek				195	503	2.58
DetachAccept (NAS)	Samsung	-	-	-	2217	3681	1.66
	MediaTek	EMMStatus	-	EMMStatus	205	1167	5.69
DetachRequest (NAS)	Samsung	DetachAccept	-	DetachAccept	2218	4776	2.15
	MediaTek				205	4434	21.63
GUTIReallocationCommand (NAS)	Samsung	GUTIReallocationComplete	-	GUTIReallocationComplete	2218	4837	2.18
	MediaTek	-	-	-	205	921	4.49
IdentityRequest (NAS)	Samsung	IdentityResponse	-	IdentityResponse	2219	4796	2.16
	MediaTek	-	-	-	205	728	3.55
SecurityMode (NAS)	Samsung	-	-	-	2217	3426	1.55
	MediaTek	SecurityModeReject	-	SecurityModeReject	205	1278	6.23
DeactivateEPSBearerContextRequest (NAS)	Samsung	DeactivateEPSBearerContextAccept	-	DeactivateEPSBearerContextAccept	2217	6102	2.75
	MediaTek				205	2934	14.31
ESMInformationRequest (NAS)	Samsung	ESMStatus	-	ESMStatus	2218	6041	2.72
	MediaTek	-	-	-	205	2749	13.41

the parameterisation of one or multiple DL packets that are sent to the UE, and a series of checks on its UL responses.

For each test, TS36.523-1 provides information on the protocol layers tested, and the state that the UE is expected to be in, prior to the test being started. As implementing all tests specified in the 6447 pages of TS36.523-1 is unfeasible, we filtered the conformance tests for those that (1) concern the RRC and NAS layers, the two primary protocols of interest to us, (2) require the UE to be in a connected state, the primary state of interest to us, and (3) do not require the UE to be in an engineering mode not accessible to regular users and thus irrelevant from a security perspective.

We implemented five of the conformance tests matching these criteria in Python. Our implementation generates the DL packets and encodes them using ASN.1 and the NAS tabular encoding. We then observe UL messages sent by the emulated basebands in FIRMWIRE and BASEBRIDGE by attaching to their GSMTAP interfaces. All UL packets are validated according to the rules defined in the conformance test. An overview of the results is contained in Table 3.

The "**Test procedure to check RRC_CONNECTED state**" test sends a UECapabilityEnquiry message. The UE should only respond whenever the RRC connection procedure is already completed and the UE is not in an idle state. Both emulated basebands pass this test in BASEBRIDGE, but fail in FIRMWIRE due to a missing UL response.

The "**RRC connection reconfiguration / Radio resource reconfiguration**" test first sends a DL RRC ConnectionReconfiguration packet in which the network instructs the UE to setup logical channels, configures a power headroom report, and specifies certain signal strengths to be used. It then expects an RRC ConnectionReconfigurationComplete in response, and subsequently performs a test procedure to check the `RRC_CONNECTED` state of the baseband.

This test fails for both emulated basebands in BASEBRIDGE and, curiously, for the physical Samsung UE. A trace analysis reveals that for the MediaTek baseband, the missing `L2COPRO` peripheral causes this issue, similar to what we observed for the RRC ConnectionReconfiguration in our test packets. For Samsung, even if not in-line with the specification, the behaviour of the UE emulated with BASEBRIDGE matches the physical UE's implementation.

In the "**MT-SMS over SGs / Active mode**" test, a Short Message Service (SMS) text message is sent from the network to the UE. The test then verifies if the UE replies with two acknowledgments, a *CP-ACK* acknowledging that the SMS has been received, and an *RP-ACK* acknowledging that the SMS was successfully decoded, forwarded to the Application Processor (AP) running, e. g., Android, and acknowledged by this processor. Only after this, the network replies with another acknowledgment (the *RP-ACK*). Without manual intervention, both basebands emulated via BASEBRIDGE do not pass this test case. The MediaTek baseband emulated via BASEBRIDGE responds with a CP-ACK but the RP-ACK is missing for two reasons. First, we had to implement, in three lines of code, a hook that replies to the baseband's attempt to forward the SMS to the AP with the appropriate AT-Command. Second, we had to restore three more memory chunks. We found these chunks using the *Dynamic Memory Restoring* algorithm after disabling its grouping step (Section 4.2). This usually leads to many addresses being restored incorrectly, but by applying it at a stage where most of the memory addresses are already restored, this discovers the missing addresses for this conformance test. With these adjustments the MediaTek baseband successfully passes the conformance test.

The Samsung baseband succeeds to send a CP-ACK but fails to reply with an RP-ACK. Upon inspecting the logs, we notice the same issue as we encountered in the MediaTek

TABLE 3. CONFORMANCE TESTING RESULTS OF BASEBRIDGE. ✓: TEST PASSED, ✓: TEST PASSED WITH MANUAL INTERVENTION, ✗: TEST FAILED.

Protocol	Name	Reference	Vendor	FIRMWIRE # Packets	BASEBRIDGE # Packets	BASEBRIDGE Passed?
RRC	Test procedure to check RRC_CONNECTED state	TS 36.508 6.4.2.3	Samsung MediaTek	1/2 1/2	2/2 2/2	✓ ✓
RRC	RRC connection reconfiguration / Radio resource reconfiguration	TS 36.523-1 8.2.2.1	Samsung MediaTek	1/4 1/4	1/4 1/4	✗ ✗
NAS	MT-SMS over SGs / Active mode	TS 36.523-1 11.1.2	Samsung MediaTek	1/4 1/4	2/4 2/4	✗ ✓
NAS	Identification procedure / IMEI requested	TS 36.523-1 9.1.4.2	Samsung MediaTek	1/2 1/2	2/2 2/2	✓ ✓
NAS	Identification procedure / IMEISV requested	TS 36.523-1 9.1.4.2	Samsung MediaTek	1/2 1/2	2/2 2/2	✓ ✓

baseband: the communication with the AP. However, we were unable to identify a fix via manual intervention.

The "**Identification procedure / IMEI requested**" and "**Identification procedure / IMEISV requested**" conformance tests send a DL NAS Information Request packet. This packet requests the UE to respond with its International Mobile Equipment Identity (IMEI), or with its IMEISV – an extended IMEI containing the software version installed on the UE. The tests then verify whether the UE responds accordingly with a NAS Identity Response, containing the requested identifier. All basebands emulated via BASEBRIDGE pass this test, with no response via FIRMWIRE.

Our evaluation on both our self-created broad set of test messages and the 3GPP compliance tests shows that BASEBRIDGE significantly increases the capabilities of emulated basebands. Most importantly, we can achieve *interactivity*, that is, the emulated basebands respond to DL packets with UL packets, often matching the behavior of the physical device. If we see deviations from the behavior of physical basebands, we are able to examine them. As we have seen, these deviations usually lead to missing UL responses due to early termination of packet processing, and security researchers are not mislead into thinking a physical UE would exhibit a behavior that is only present in the emulated version. Moreover, even in cases where missing peripherals prevent progress, we achieve a significantly higher number of executed basic blocks than FIRMWIRE. Our approach is therefore a significant improvement over the state of the art, even in cases where additional emulation of peripherals or further manual intervention is strictly required.

6.2. Fuzz Testing with BASEBRIDGE

To further show the effectiveness of BASEBRIDGE, we compare our tool against FIRMWIRE in 2 fuzzing evaluations. In the first evaluation, we use plain AFL++ [11] for both BASEBRIDGE and FIRMWIRE. For the second evaluation we also conduct experiments for both BASEBRIDGE and FIRMWIRE, but implement a custom grammar-aware mutator to increase the probability of a correctly decoded RRC packet, as motivated below. While such an approach has been discussed in previous research on OTA fuzzing [12], [36], we are not aware of any publicly available implementation of such a mutator.

Experiment setup. For each setup, we run 15 24-hour trials on a server with 2 Intel(R) Xeon(R) Gold 5320 CPUs (26 physical cores, 52 logical cores each). To minimize scheduling noise, we instruct AFL++ (version $4.22a$) to use an interleaved pattern when assigning cores to our fuzzing runs. We run 30 fuzzing experiments in parallel, 15 on each CPU. We use the same modkit harnesses as described in Section 6, as well as a *restore list file* obtained from replaying the DL messages from Subsection 6.1. All coverage metrics are measured after the baseband is booted, and thus do not include coverage that is unrelated to the fuzzing input.

Custom mutator. We utilize AFL++'s Python interface for custom mutators. On every mutation, an RRC packet type is randomly chosen and an empty packet of this type is created. It is then filled by recursively traversing the Abstract Syntax Notation One (ASN1) definitions of RRC and generating random values for every component. Although NAS is transported via RRC, NAS messages are not subject to this generation algorithm as NAS packets do not use ASN1 encoding per the specification. Instead, NAS messages are generated completely randomly. Our implementation could easily be adopted to other protocols that facilitate ASN1.

Results. Figure 3a and Figure 3c show the coverage changes within the first experiment, using the standard AFL++ mutator. For the Samsung baseband, BASEBRIDGE outperforms FIRMWIRE on average by 40149 basic blocks, an improvement by a factor of 2.3. For the MediaTek baseband, we see that BASEBRIDGE outperforms FIRMWIRE by 18729 basic blocks, resulting in an improvement by a factor of 4.1.

Similarly Figure 3b and Figure 3d illustrate the coverage differences in our second experiment, based on our custom mutator. BASEBRIDGE outperforms FIRMWIRE on average by 35852 basic blocks for Samsung, an improvement by a factor of 3.1. In the same setup targeting MediaTek, BASEBRIDGE outperforms FIRMWIRE on average by 11706 basic blocks, a more than 5x improvement. We speculate that the custom mutator leads to larger relative improvements, as most executions will explore a significant amount of state-dependent functionality, while the AFL++ mutator explores more state-independent parsing related error cases, slightly decreasing the relative gap between both approaches.

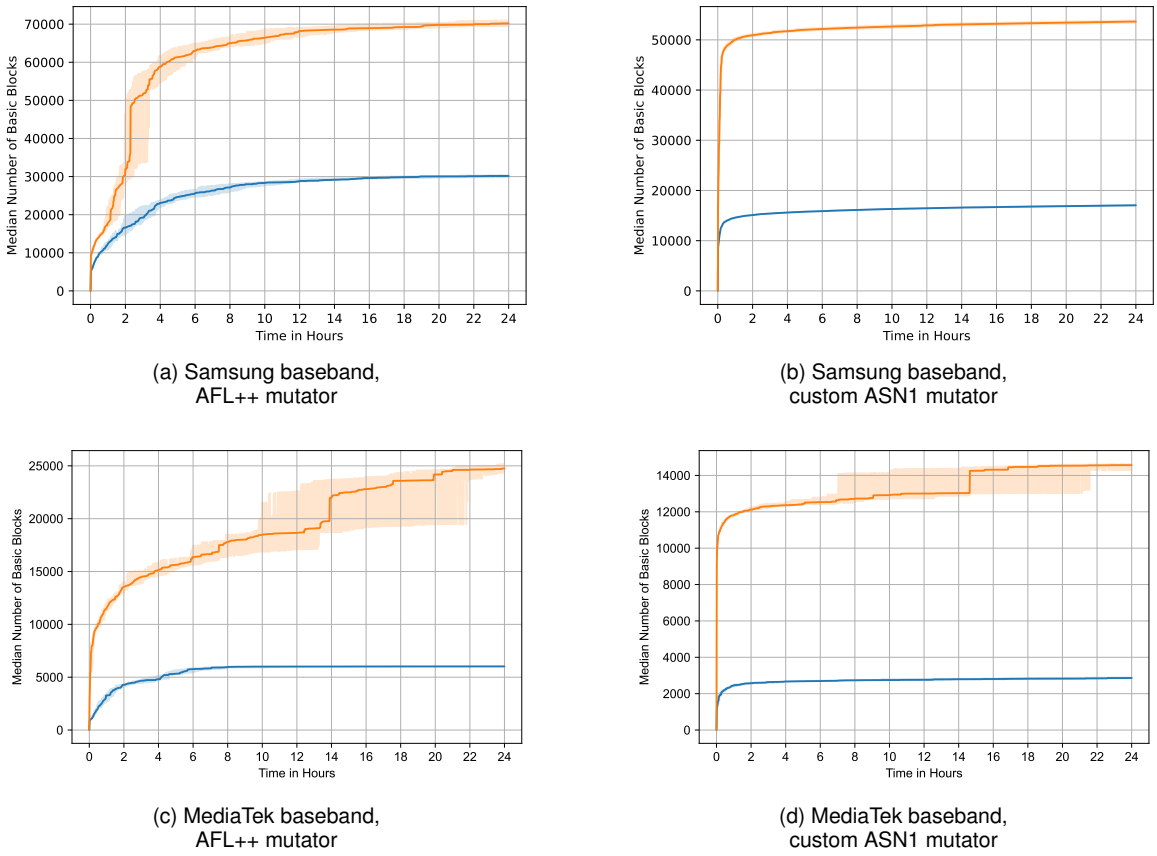

Figure 3. Results of fuzzing runs showing median and a 95% confidence interval.— FIRMWIRE— BASEBRIDGE

We also calculated the per-task RRC coverage, following the proxy measurements established by FIRMWIRE, as shown in Table 4. Here, we see even more significant differences compared to the plots. For MediaTek, BASEBRIDGE covers up to 22.5x the number of RRC related basic blocks compared to FIRMWIRE for the MediaTek RRC task. The difference is slightly less significant using our custom mutator, as the gap between FIRMWIRE and BASEBRIDGE shrinks to a factor of 18.9x. For Samsung, we see that BASEBRIDGE covers 9.7 times as many basic blocks of the RRC task using the default mutator and 9.0 times as many blocks using the custom mutator.

Our results indicate that the AFL++ greybox mutator generally outperforms our blackbox grammar-aware mutator. More broadly, this means that OTA-fuzzing, which is generally blackbox, will likely suffer from the same limitation, compared to greybox fuzzing enabled by emulation. Our results also show that, using BASEBRIDGE, fuzzing coverage of functionality closely related to the received messages (e.g., the RRC task) is improved over-proportionally, while coverage of functionality not directly included within the RRC processing is still significantly improved. Overall, our results thus demonstrate the tremendous coverage improvements that BASEBRIDGE enables across both mutation strategies and the entire emulated firmware.

TABLE 4. COVERAGE, MEASURED IN BASIC BLOCKS, OF THE RRC TASK THROUGHOUT OUR FUZZING EXPERIMENTS.

	FIRMWIRE		BASEBRIDGE	
Mutator	**AFL++**	**Custom**	**AFL++**	**Custom**
Samsung	2446	2386	23846	21381
MediaTek	374	353	8403	6688

TABLE 5. DISCOVERED VULNERABILITIES.

	Message	New find	CVE ID	Severity
Samsung	Reconfiguration (RRC #2)	✓	*outstanding*	medium
	Reconfiguration (RRC #3)	✓	*outstanding*	high
	EMMInformation (NAS)	✗	CVE-2024-39343	high
	ModifyEPSBearerContextReq. (NAS)	✗	CVE-2023-21517 [28]	high
MediaTek	ConnectionRelease (RRC #1)	✓	CVE-2024-20154	critical
	DLTransport (NAS #1)	✓	CVE-2024-20149	medium
	TrackingAreaUpdateAcc. (NAS #2)	✓	CVE-2024-20150	medium
	EMMInformation (NAS #3)	✗	CVE-2024-20039	high

6.3. Discovered Vulnerabilities

We also utilized BASEBRIDGE for long-lasting fuzzing campaigns with the goal of vulnerability discovery, again focusing on RRC and NAS messages using both the standard AFL++ mutator and our custom mutator. To aid these fuzzing campaigns, we periodically re-ran the *Dynamic*

```
RRCConnectionRelease
    releaseCause: 1 (other)
    idleModeMobilityControlInfo:
        freqPriorityListGERAN [
            freqsPriorityGERAN:
                carrierFreqs:
                    startingARFCN: 4
                    followingARFCNs:
                        variableBitMapOfARFCNs: 0xA000
                cellReselectionPriority: 6
            freqsPriorityGERAN:
                carrierFreqs:
                    startingARFCN: 8
                    followingARFCNs:
                        variableBitMapOfARFCNs: 0x01
                cellReselectionPriority: 2
        ]
        t320: min180
```

Listing 2. Downlink RRC Connection Release packet using the `variableBitMapOfARFCNs` feature exploited in RRC #1.

```
1  arfcn = startingArfcn;
2  for (i = 0; byteIdx < bitMapLen; byteIdx++) {
3    bitIdx = 7;
4    do {
5      arfcn = arfcn + 8 - bitIdx;
6      if ((1 << (bitIdx & 0x1f) & arfcnBitmap) != 0) {
7        frequencies->priorityList[arfcnIdx].arfcn = arfcn;
8        frequencies->priorityList[arfcnIdx].xyz = xyz;
9        arfcnIdx = arfcnIdx + 1 & 0xff;
10       if (arfcnIdx == 128) break;
11     }
12     bitIdx--;
13   } while (bitIdx != -1);
14 }
```

Listing 3. Reverse-engineered excerpt of ARFCN bitmap parsing in MediaTek firmware.

Memory Restoring algorithm to improve emulation accuracy based on generated inputs from previous fuzzing runs, potentially revealing functionality not targeted by the DL packets from Table 6.

Overall, our campaign found 8 vulnerabilities, with 5 previously unknown, as shown in Table 5. Below, we provide case studies of selected discovered vulnerabilities, and provide details of the remaining previously unknown vulnerabilities in Appendix B.

RRC #1: Connection Release. When fuzzing the MediaTek baseband firmware, we found that the firmware does not correctly handle a combination of a timer value and a cell reselection priority in the RRC ConnectionRelease DL packet. Under normal operation, this feature is used to force a physical UE to disconnect from the current network and try to connect to another cell on another frequency, based on an Absolute Radio Frequency Channel Number (ARFCN) priority list supplied in this packet [4]. We found that when supplying a certain frequency configuration, the firmware will perform an out-of-bounds write to a variable on the stack when storing the updated frequency priorities.

Specifically, this involves the `startingARFCN` and `variableBitMapOfARFCNs` fields inside `idleModeMobilityControl`. These fields are used to pass a priority list of several ARFCNs in a compressed way: If the bit at offset x in the `variableBitMapOfARFCNs` is 1, the frequency channel represented by the ARFCN (x + `startingARFCN`) (modulo 1024) is added to the reselection priority list. For example, the packet shown in Listing 2 configures a reselection priority of 6 for 3 different ARFCNs – 4 (the `startingARFCN`), 5 (because the first bit in `0xA000` is 1) and 7 (because the third bit in `0xA000` is 1) – and a priority of 2 for ARFCNs 8 and 16.

An excerpt from the relevant part of MediaTek's firmware is shown in Listing 3. This parser copies all prioritized ARFCNs into the `priorityList` array, which can store a maximum of 128 different ARFCNs. The outer for-loop (line 2) iterates over the bytes in `variableBitMapOfARFCNs`, while the inner while-loop (line 4) iterates over the individual bits to compute the corresponding ARFCN (line 5). When a bit in the bitmap is 1 (line 6), the ARFCN is stored in the `priorityList` (line 7). Once 128 different ARFCNs have been decoded, only the inner loop is terminated (line 10); the outer loop incorrectly continues with the next byte. As such, as long as there is one more byte in the bitmap after this point, the parser will overflow the `priorityList` array.

The next element in the `frequencies` struct after the `priorityList` is a length value `frequencies->len`, followed by a buffer `frequencies->buf`; these will be overwritten when an overflow occurs. During processing of the `frequencies` struct, `len` bytes of `buf`'s content are copied to a fixed-sized stack-based buffer. Since both `len` and `buf` are attacker-controlled after the overflow, they can cause an overflow from the stack-based buffer into the return address register stored on the stack. Our fuzzer found this issue by generating a packet that overflowed the stack-based buffer with zeroes, eventually setting the return address register and program counter to 0 and causing a crash.

This vulnerability could potentially lead to remote code execution, and since the RRCConnectionRelease packet is accepted by basebands prior to enabling integrity protection or encryption, it is exploitable without control of the cellular network by broadcasting packets via an SDR. This vulnerability was previously unknown, and MediaTek assigned it *CVE-2024-20154* with critical severity.

RRC #2: Reconfiguration. This vulnerability is discovered by our Samsung LTE RRC fuzzer whenever a malformed RRC Reconfiguration message, containing a Maximum Context Identifier (CID) field with a value greater than 16 is sent over the DCCH channel (Listing 4) to the baseband.

According to the specification [1], the baseband should be able to handle small (`<=16`) and large CIDs (`>16`), indicated by the `maxCID` value. Upon receiving an RRC Reconfiguration message with a `pdcp-Config` field, the baseband allocates a heap buffer (`buf`) of 66 bytes, independent of the CID value. We suspect that, within the 66 bytes, 22 bytes are used to store the enabled CIDs (bytes 3-18), and 2 bytes are used to store the `maxCID` value (bytes 19 and 20). Upon further processing, the baseband executes a *for loop*, intended to write the value 1 `maxCID` times, starting at `buf + 3` (the CIDs). Whenever an attacker sends an RRC Reconfiguration message containing a `maxCID` value

```
rrcConnectionReconfiguration
    (..)
    radioResourceConfigDedicated:
        drb-ToAddModList: [
            DRB-ToAddMod
                drb-Identity: 1
                pdcp-Config
                    discardTimer: infinity (7)
                    headerCompression: rohc (1)
                        rohc
                            maxCID: 5258
        ]
```

Listing 4. Downlink RRC Connection Reconfiguration packet using the large `maxCID` feature exploited in RRC #2.

```
1  // &packet->encoded_nw_name is 255 bytes or less
2  byte nw_name_unpacked [260];
3  if (nw_name_encoding == 0) { // 7 bit encoding
4      csmss_gsm7_unpack(
5          nw_name_unpacked,
6          &packet->encoded_nw_name,
7          nw_name_len, 0
8      );
9  } else { // 8 bit encoding, no unpacking needed
10     memcpy(nw_name_unpacked,
11         &packet->encoded_nw_name,
12         nw_name_len);
13 }
```

Listing 5. Handling of the network name EMM Information in MediaTek firmware.

greater than 62, the *for loop* causes an out-of-bounds heap write to `buf`. Ultimately, upon freeing `buf`, the baseband crashes with a `PAL_MEM_GUAD_CORRUPTION`. Google has assigned a Medium severity to this vulnerability.

NAS #3: EMM Information. When fuzzing the MediaTek baseband firmware, we found that the firmware does not reserve enough space in memory for the network names carried in EMM Information NAS packets. Network names allow the user to identify to which Mobile Network Operator they are connected. They can be sent using various encodings; one is a 7-bit per character form of ASCII, which the baseband needs to unpack to 8-bit ASCII for internal use.

Listing 5 shows an abbreviated version of MediaTek's implementation of this unpacking. The baseband firmware allocates 260 bytes on the stack for the 8-bit encoded network name, in `nw_name_unpacked`. If the received network name in `packet->encoded_nw_name` is using 7 bits per character (line 3), then it is decoded via `csmss_gsm7_unpack(out, in, size)`. The NAS parser, which we exclude in the sample for brevity, limits the `encoded_nw_name` variable to 255 bytes. This means that `csmss_gsm7_unpack` will write a theoretical maximum of 255 * 8/7 = 292 bytes to `nw_name_unpacked`, overflowing the allocated stack space and overwriting the stored return address register. Once the function returns, the program counter is set to that value.

We have verified this vulnerability using the NAS packet in Listing 6. The 7-bit encoded text payload corresponds to 268 bytes containing `0x3d` when decoded to 8-bit ASCII. `csmss_gsm7_unpack` will then overflow into 2 bytes of the stored return address register – overwriting the last byte

```
Protocol discriminator: 0x7 (EPS mobility management)
Message type: 0x61 (EMM Information)
    Element ID: 0x45 (Network Name - Short Name)
    Length: 237
    Spare bits: 6
    Coding Scheme: 0
    Text: 0xbd 0x5e 0xaf 0xd7 0xeb 0xf5 0x7a
       (repeated for 235 bytes)
```

Listing 6. Downlink NAS message to trigger NAS #3.

with `0x3d` and the previous byte with a null terminator (`0x00`). The first part of the stored value remains `0x909f`. This diverts execution flow to the function at `0x909f003d`, which prints several log messages absent from normal executions, allowing us to confirm it was reached.

This vulnerability was previously known by MediaTek and was assigned *CVE-2024-20039*, but had not been patched in the firmware version we were testing. The external researcher who originally found this vulnerability kindly provided us with their vulnerability report which revealed that they were not aware that this is a remote code execution vulnerability, as their OTA approach did not provide sufficient introspection to distinguish between an attacker controllable program counter, and a mere denial of service. This highlights the advantages of stateful testing under emulation, as provided by BASEBRIDGE.

6.3.1. Over-The-Air reproduction. To ensure our fuzzing results are also applicable to real devices, we reproduced LTE NAS #1, #2, #3 and LTE RRC #1, #2, #3 over-the-air.

For vulnerabilties affecting MediaTek basebands, we used a Samsung A41 and a Poco M4 Pro 5G[4], and for those affecting Samung basebands, the Google Pixel 6 and 8[5]; the Galaxy S10e was already end of life at the time of testing.

Setup. For the over-the-air reproduction, we used an SDR (e. g., BladeRF or USRP) connected to a laptop. We used a Faraday cage to avoid interference with legitimate cellular networks. For the basestation software, we used srsRAN 4G (release 23.11) built from source. We modified srsRAN to send our payload after the RRC and NAS connections have been established and integrity protection has been enabled on both layers. This is consistent with the fact that the crash dumps (and thus the state) used by BASEBRIDGE were captured from UEs in connected state.

We confirmed a Cellular Processor (CP) crash visually by observing the loss of connection displayed on the mobile device's screen, as well as systematically by either analyzing the mobile device's radio logs (Android Debug Bridge (ADB) Logcat) or by flashing a User Debug Android image on the Pixel devices and observing crash dump files.

Results. We successfully reproduced all tested vulnerabilities against at least one physical UE resulting in the expected crash. As discussed above, we further verified the LTE NAS #3 RCE vulnerability by directing execution to a normally

4. FW versions `A415FXXS2DWA2` and `V816.0.1.0.TGBEUXM`
5. FW versions `AP2A.240905.003.F1` and `AP3A.241105.007`

unused function, and checking the baseband logs to confirm the presence of log messages created by this function.

7. Discussion

Applicability to other cellular generations. Although our experiments focus on LTE implementations, we believe that BASEBRIDGE's approach seamlessly scales to protocol stacks of other cellular generations. Since no baseband emulator currently supports firmware of a 5G baseband, we have not evaluated BASEBRIDGE on the 5G protocol and we argue that adding support for 5G UEs is primarily an engineering effort beyond the scope of this paper.

However, we tested BASEBRIDGE's approach for Samsung's 2G GSM and GPRS stacks, inspired by concurrent work on reverse-engineered state for fuzzing purposes based on significant manual effort [14], [15]. Using BASEBRIDGE we were able to replicate the findings of this work, including complex intra-stack communication and re-discovery of vulnerabilities with very little additional work. Hence, we think that once emulators for other protocol stacks, including 5G, are available, BASEBRIDGE will directly be applicable and tremendously aid dynamic analysis.

Applicability to other firmware images. Although our experiments focus on firmware images of a Samsung Galaxy S10e (Samsung) and a Samsung Galaxy A41 (MediaTek), BASEBRIDGE's approach is scalable to other firmware images. As a proxy, we tested BASEBRIDGE's approach on the firmware images of a rooted Motorola One Vision and a non-rooted Samsung Galaxy S9[6], both using Samsung basebands. Using BASEBRIDGE, we successfully replayed the DL messages from Table 6, matching the response in BASEBRIDGE to the physical UEs. Overall, the Motorola One Vision produces a 2.04x increase, while the Samsung Galaxy S9 produces a 2.02x increase in coverage with BASEBRIDGE compared to FIRMWIRE. The Motorola One Vision also shows our ability of retrieving a crash dump from a UE that does not expose a debug menu, leveraging an `IOCTL` command instead.

Interestingly, and in contrast to the previously tested Samsung S10e, both UEs respond with a NAS SecurityModeAccept upon receiving a NAS SecurityMode DL message, even after the security mode has already been configured. This is true for both, the physical UE and the emulated baseband in BASEBRIDGE. This shows that different Samsung basebands can behave differently when receiving the same DL message, and that BASEBRIDGE is capable of maintaining these differences in emulation.

Restoring different states. The experiments above are based on memory dumps obtained from physical basebands in the `RRC_CONNECTED` state, as this state offers the largest amount of functionality. Memory dumps can also be obtained in other states defined by the cellular specification, by following the specification's procedures to reach those states on the physical baseband and using one of the methods described in Subsection 4.1 to obtain the dump.

However, generating memory dumps in more fine-grained states (such as in the middle of a authentication procedure) is not always possible. These states are often implementation-specific, requiring manual effort to determine which states exist and how to reach them. Many such states are also transient – the baseband will leave them within a few milliseconds, making it infeasible to manually trigger a memory dump. As such, it may not be practical to obtain memory dumps for these states, which limits BASEBRIDGE's capabilities in these scenarios.

An alternative approach for reaching some of these fine-grained states using BASEBRIDGE is to start emulation by restoring a well-defined state, and then send a sequence of packets to the emulator to reach the desired fine-granular state. Although limitations, such as missing peripherals, mean that this approach may not always work as-is, our conformance test results demonstrated that BASEBRIDGE is generally capable of handling such complex procedures.

Restoring state vs improving emulation capabilities. As discussed, enabling baseband emulators to be able to reach specific states independently, without the need for restoring state from a real device would be a huge effort, requiring support for a large number of peripherals (including SIM cards, DSP logic and RF frontends). This would require tedious reverse-engineering and is likely to be highly device- and vendor-specific. However, improving emulator fidelity may well improve BASEBRIDGE's coverage, especially if applied in a targeted fashion as a response to encountered issues. For example, peripheral emulation could be improved by researchers when incomplete emulation is clearly preventing progress on a certain path, such as the issue we encountered with the MediaTek firmware due to the missing emulation of the `L2COPRO` peripheral.

This approach seems likely to provide an ideal compromise, allowing researchers to continue to benefit from restored state without the need to invest the effort required for complete emulation. Additionally, BASEBRIDGE's improved dynamic analysis capabilities may ease the reverse-engineering required for such improvements.

BASEBRIDGE beyond fuzzing. We have shown that BASEBRIDGE significantly increases the amount of baseband functionality reachable by emulator-based fuzzing campaigns, allowing such testing to be done using a scalable emulation approach rather than brittle, expensive and non-scalable OTA testing methods. We believe our approach will enable similar scaling in other areas beyond fuzzing, especially since BASEBRIDGE enables interactivity complete with UL responses. For example, BASEBRIDGE could provide similar benefits for state-of-the-art UE security testing techniques [7], [34] which currently rely on OTA testing.

6. FW versions `KANE_RETEU_11_RSA31.Q1-48-36-23` and `G960FXXS7CSJ3`

8. Related Work

Protocol security. Several previous works propose model-based methods to discover security flaws in cellular protocols or their implementations. 5GReasoner [19] proposes a formal verification model for 5G, and LTEInspector [18] developed a symbolic model checker to discover security flaws in the LTE protocol. Basespec [23] combines static analysis with symbolic execution to find mismatches between the specification's NAS message structure and the message structure assumed by the baseband's internal message parsers. BaseComp [22] proposes a semi-automatic approach by combining static analysis with symbolic execution to discover discrepancies between the specification and the baseband's NAS integrity protection implementation.

Other approaches to find protocol-related security vulnerabilities involve OTA testing [6], [21]. 5GBaseChecker [42] and DIKEUE [20] utilize automata learning to model the cellular protocol's behavior as a Finite State Machine (FSM). Protocol security flaws are found by analyzing inputs that result in deviating FSMs through differential testing. DoLTEst [34] tests COTS devices OTA against a deterministic oracle when handling negative RRC and NAS messages to reveal flaws in the implementation of LTE protocols. Other work deduces implementation security flaws via specification analysis [7], [8], [25] and then verify them OTA. Similarly, recent industry work [41] developed an automated framework to identify security policy violations in 5G UE implementations. With BASEBRIDGE, we enable much more introspection into the baseband's internal state, while incorporating the stateful advantages of OTA testing.

Cellular security. A common tool to uncover implementation flaws in basebands is static analysis. BVFinder [31] proposes a semantic-enhanced vulnerability detector using static taint analysis, while BaseMirror [29] proposes a static binary analysis tool to automatically reverse baseband commands from Radio Interface Layer (RIL) binaries.

Contrary to static analysis, Beserker [36] proposes a mutation-based fuzzer to randomly test LTE RRC and NAS messages. CovFuzz [40] implements a coverage-based fuzzer to fuzz the LTE and 5G - NAS attach message. Similarly, recent industry work such as 5GHoul [13] discovered several implementation flaws in MediaTek and Qualcomm basebands. Unlike existing static analysis-based or OTA-based approaches, BASEBRIDGE leverages the advantages of emulation-based testing.

Recently, rehosting of baseband firmware has drawn a lot of attention. In addition to FIRMWIRE [17], BaseSafe [32] combines partial system emulation with coverage-based fuzzing to allow for in-depth introspection into the baseband's internal code with fast, off-device COTS UE fuzzing. Similarly, recent work from industry developed closed-source emulators for baseband firmware [26], [27], [39], [44]. Additionally, SIMurai [30] offers a software implementation of a SIM capable of communicating with an emulated baseband through FIRMWIRE, allowing to fuzz SIM related functionality in the baseband. Unlike existing rehosting approaches, BASEBRIDGE leverages the advantages of stateful testing.

Instead of analyzing the UE, other research is tailored towards the core network. For example, RANsacked [5] proposes a grammar-based fuzzer to generate payloads triggering implementation errors in the 5G core network, while others [12], [24] also target the BTSs using similar fuzzing approaches.

Stateful analysis & state transfer. Research on rehosting, especially the field of hardware-in-the-loop rehosting, often relies on transfering state from a physical device to an emulator [10], [43]. The closest rehosting study to BASEBRIDGE is Frankenstein [37] which emulates and fuzzes Bluetooth firmware based on a physical device's memory dump. However, in contrast to BASEBRIDGE, prior approaches could rely on restoring and emulating the *full* dump, whereas BASEBRIDGE needs to selectively and iteratively restore state due to the much higher complexity of baseband firmware.

Outside the domain of rehosting, AFLnet [35] proposes to incorperate the server's state as feedback mechanism in fuzzing servers. The server's state is deduced from the server's response code to incoming message sequences.

9. Conclusion

We have demonstrated that BASEBRIDGE is practical by implementing a prototype supporting two completely different baseband firmwares from two different vendors. In both cases we obtain significantly increased coverage compared to existing approaches, as well as improved accuracy of the emulated baseband as reflected in the response to test messages. This demonstrates the importance of state in baseband firmware, as well as the benefits of our approach which restores this state from a real device. The value of these improvements is confirmed by the new vulnerabilities we found when fuzzing protocols which have already been extensively covered by previous work.

Acknowledgments

We thank our anonymous reviewers for their valuable comments and suggestions. This work was supported by the German Federal Office for Information Security (FKZ: Pentest-5GSec - 01MO23025B), the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and by NWO through NWA ORC "INTERSECT" and project 20475 "P6". For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

References

[1] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA); Packet Data Convergence Protocol (PDCP) specification. TS 36.323, 3rd Generation Partnership Project (3GPP), 01 2010.

[2] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Packet Core (EPC); Common test environments for User Equipment (UE) conformance testing. TS 36.508, 3rd Generation Partnership Project (3GPP), 06 2011.

[3] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Packet Core (EPC); User Equipment (UE) conformance specification; Part 1: Protocol conformance specification. TS 36.523-1, 3rd Generation Partnership Project (3GPP), 06 2011.

[4] 3GPP. Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification. TS 36.331, 3rd Generation Partnership Project (3GPP), 06 2011.

[5] Nathaniel Bennett, Weidong Zhu, Benjamin Simon, Ryon Kennedy, William Enck, Patrick Traynor, and Kevin RB Butler. Ransacked: A domain-informed approach for fuzzing lte and 5g ran-core interfaces. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, 2024.

[6] Evangelos Bitsikas, Syed Khandker, Ahmad Salous, Aanjhan Ranganathan, Roger Piqueras Jover, and Christina Pöpper. UE Security Reloaded: Developing a 5G Standalone User-Side Security Testing Framework. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2023.

[7] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Baoxu Liu. Sherlock on Specs: Building LTE Conformance Tests through Automated Reasoning. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3529–3545, Anaheim, CA, August 2023. USENIX Association.

[8] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. Bookworm Game: Automatic Discovery of LTE Vulnerabilities Through Documentation Analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1197–1214, 2021.

[9] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, New York, NY, USA, 2015. Association for Computing Machinery.

[10] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[12] Matheus E. Garbelini, Zewen Shang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Towards Automated Fuzzing of 4G/5G Protocol Implementations Over the Air. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 86–92, 2022.

[13] Matheus E. Garbelini, Zewen Shang, Shijie Luo, Sudipta Chattopadhyay, Sumei, and Ernest Kurniawan. 5GHOUL: Unleashing Chaos on 5G Edge Devices. Technical report, Singapore University of Technology and Design (SUTD) and I2R, A*STAR, 2023.

[14] Dyon Goos and Marius Muench. Fuzzing GPRS Layer-2 for Fun and Profit. Hardwear.io Netherlands 2024, 2024.

[15] Dyon Goos and Marius Muench. Overcoming State: Finding Baseband Vulnerabilities by Fuzzing Layer-2. Black Hat USA, 2024.

[16] GSMA. The Mobile Economy 2023, 2023.

[17] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *Symposium on Network and Distributed System Security (NDSS)* , 2022.

[18] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Network and Distributed System Security Symposium*, 2018.

[19] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 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 '19, page 669–684, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1082–1099, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Syed Khandker, Michele Guerra, Evangelos Bitsikas, Roger Piqueras Jover, Aanjhan Ranganathan, and Christina Pöpper. ASTRA-5G: Automated Over-the-Air Security Testing and Research Architecture for 5G SA Devices. In *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '24, page 89–100, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Eunsoo Kim, Min Woo Baek, CheolJun Park, Dongkwan Kim, Yongdae Kim, and Insu Yun. BASECOMP: A Comparative Analysis for Integrity Protection in Cellular Baseband Software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3547–3563, Anaheim, CA, August 2023. USENIX Association.

[23] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. BaseSpec: Comparative analysis of baseband software and cellular specifications for l3 protocols. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Online, February 2021.

[24] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1153–1168, 2019.

[25] Daniel Klischies, Moritz Schloegel, Tobias Scharnowski, Mikhail Bogodukhov, David Rupprecht, and Veelasha Moonsamy. Instructions Unclear: Undefined Behaviour in Cellular Network Specifications. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3475–3492, Anaheim, CA, August 2023. USENIX Association.

[26] Daniel Komaromy. Basebanheimer: Now I Am Become Death, The Destroyer of Chains. Hardwear.io Netherlands, 2023.

[27] Daniel Komaromy. There will be Bugs: Exploiting Basebands in Radio Layer Two. CanSecWest, 2024.

[28] David Komaromy. CVE-2023-21517: Samsung Baseband LTE ESM TFT Heap Buffer Overflow, 2023.

[29] Wenqiang Li, Haohuang Wen, and Zhiqiang Lin. BaseMirror: Automatic Reverse Engineering of Baseband Commands from Android's Radio Interface Layer. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, 2024.

[30] Tomasz Piotr Lisowski, Merlin Chlosta, Jinjin Wang, and Marius Muench. SIMurai: Slicing Through the Complexity of SIM Card Security Research. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4481–4498, Philadelphia, PA, August 2024. USENIX Association.

[31] Yiming Liu, Cen Zhang, Feng Li, Yeting Li, Jianhua Zhou, Jian Wang, Lanlan Zhan, Yang Liu, and Wei Huo. Semantic-Enhanced Static Vulnerability Detection in Baseband Firmware. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[32] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20, page 122–132, New York, NY, USA, 2020. Association for Computing Machinery.

[33] P1 Security S.A.S. GitHub: QCSuper. https://github.com/P1sec/QCSuper, 2024. [Online; accessed May 7, 2025].

[34] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth Downlink Negative Testing Framework for LTE Devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1325–1342, Boston, MA, August 2022. USENIX Association.

[35] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465, 2020.

[36] Srinath Potnuru and Prajwol Kumar Nakarmi. Berserker: ASN.1-based Fuzzing of Radio Resource Control Protocol for 4G and 5G. In *2021 17th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 295–300, 2021.

[37] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36. USENIX Association, August 2020.

[38] Roger Piqueras Jover Sherk Chung, Stephan Chen and Ivan Lozano. Pixel's Proactive Approach to Security: Addressing Vulnerabilities in Cellular Modems. *Google Security Block (Online; Accessed 2024-11-13)*, 2024.

[39] Natalie Silvanovich. How to Hack Shannon Baseband (from a Phone). Hardwear.io USA, 2023.

[40] Ilja Siroš, Dave Singelée, and Bart Preneel. Covfuzz: Coverage-based fuzzer for 4g&5g protocols, 2024.

[41] Kai Tu and Yilu Dong. Cracking the 5G Fortress: Peering Into 5G's Vulnerability Abyss. BlackHat USA, 2024.

[42] Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain. Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3063–3080, Philadelphia, PA, August 2024. USENIX Association.

[43] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021.

[44] Xiling Gong Xuan Xing, Eugene Rodionov and Farzan Karimi. Over the Air, Under the Radar Attacking and Securing the Pixel Modem. Black Hat USA, 2023.

Appendix A. Additional Test Results

We provide the extended version of the UL responses to DL Messages, including all used messages and conformance tests in Table 6.

Appendix B. Additional Vulnerability Details

RRC #3: Reconfiguration. This vulnerability is discovered by our Samsung LTE fuzzer, whenever a malformed RRC Reconfiguration message is sent over the DCCH channel. Upon receiving such a message, the baseband allocates a heap block of 1208 zero-bytes to store UL related radio configurations. Later, the baseband retrieves the `pucch-ConfigDedicated-v1370` IE and copies 20 bytes into the allocated heap block. Eventually, the baseband uses the first byte from this block as the size used for a memory copy operation from an offset in our heap block to a stack buffer. From introspection, we suspect that an attacker does not control the heap buffer bytes copied into the stack buffer, and these bytes appear to be 0. Whenever the baseband tries to return from this parsing function, the baseband will crash in the `RESET` function after a program counter value of 0 is popped from the stack. Even though, the specification specifies that this message should only be accepted in the `RRC_CONNECTED` state [4], we verified that this vulnerability can be reached before the over-the-air security security establishment. Google has assigned a high severity to this vulnerability.

NAS #1: DL Transport. When fuzzing the MediaTek baseband firmware, we found that the firmware does not handle empty Downlink NAS Transports appropriately. When receiving a DL NAS Transport packet, the baseband allocates a buffer in memory. Whenever the size of the payload is zero, the baseband allocates a buffer of length zero. This fails with an assertion, which leads to a forced crash. The vulnerability was previously unknown, has been assigned *CVE-2024-20149* and was rated as medium severity by MediaTek, as it allows an attacker to disable all connectivity of a physical UE over the air, until the baseband is restarted.

NAS #2: Tracking Area Update (TAU) Accept. When fuzzing the MediaTek baseband firmware, we found that the firmware does not correctly validate the length of Tracking Area Identity (TAI) lists carried in TAU Accept packets. TAI lists consist of multiple partial TAI list elements. Whenever the sum of the lengths of the partial lists is greater than the signaled length of the containing TAI list, parts of the transferred list are written out of bounds during decoding. This out of bounds write overwrites a pointer in a control structure. This pointer is immediately de-referenced and read from in the next step of the processing chain, allowing an adversary to turn this into an arbitrary read. We did not identify a channel allowing us to disclose the results of this read. However, by supplying an invalid memory address, an adversary can use this vulnerability to crash the baseband. The issue was previously unknown, has been assigned *CVE-2024-20150* and was rated as medium severity by MediaTek, as it – at the bare minimum – allows an attacker to disable all connectivity of a physical UE over-the-air until the baseband is restarted.

TABLE 6. UL RESPONSES TO DL MESSAGES (EXTENDED).

Message	Vendor	Expected Response	Reponse FIRMWIRE	Response BASEBRIDGE	BB_{cov} FIRMWIRE	BB_{cov} BASEBRIDGE	δ_{BB}
RRCConnectionRelease (RRC)	Samsung	-	-	-	3083	5459	1.77
	MediaTek				206	3614	17.54
RRCConnectionReconfiguration (RRC)	Samsung	RRCConnectionReconfigurationComplete	-	RRCConnectionReconfigurationComplete	3859	6767	1.75
	MediaTek			-	220	1757	7.99
SecurityModeCommand (RRC)	Samsung	SecurityModeFailure	-	SecurityModeFailure	2210	3130	1.42
	MediaTek				203	468	2.31
UECapabilityEnquiry (RRC)	Samsung	UECapabilityInformation	-	UECapabilityInformation	2226	3885	1.75
	MediaTek				214	2048	9.57
CounterCheck (RRC)	Samsung	CounterCheckResponse	-	CounterCheckResponse	2235	3870	1.73
	MediaTek				202	556	2.75
CSFBParameterResponseCDMA2000 (RRC)	Samsung	-	-	-	2226	2170	0.97
	MediaTek				168	267	1.59
DLDedicatedMessageSegment-r16 (RRC)	Samsung	-	-	-	1836	1489	0.81
	MediaTek				151	250	1.66
HandoverFromEUTRAPreparationRequest (RRC)	Samsung	-	-	-	2164	2867	1.32
	MediaTek				161	260	1.61
LoggedMeasurementConfiguration-r10 (RRC)	Samsung	-	-	-	2200	2136	0.97
	MediaTek				246	357	1.45
MobilityFromEUTRACommand (RRC)	Samsung	-	-	-	2231	3193	1.43
	MediaTek				207	1899	9.17
UEInformationRequest-r9 (RRC)	Samsung	UEInformationResponse-r9	-	UEInformationResponse-r9	2325	2990	1.29
	MediaTek				195	503	2.58
AttachAccept (NAS)	Samsung	-	-	-	2227	17551	7.88
	MediaTek				205	765	3.73
AttachReject (NAS)	Samsung	-	-	-	2218	3930	1.77
	MediaTek				205	728	3.55
AuthenticationReject (NAS)	Samsung	-	-	-	2218	23868	10.76
	MediaTek				205	728	3.55
AuthenticationRequest (NAS)	Samsung	-	-	-	2218	3918	1.77
	MediaTek				205	728	3.55
DetachAccept (NAS)	Samsung	-	-	-	2217	3681	1.66
	MediaTek	EMMStatus	-	EMMStatus	205	1167	5.69
DetachRequest (NAS)	Samsung	DetachAccept	-	DetachAccept	2218	4776	2.15
	MediaTek				205	4434	21.63
EMMInformation (NAS)	Samsung	-	-	-	2218	4483	2.02
	MediaTek				205	1650	8.05
EMMStatus (NAS)	Samsung	-	-	-	2218	3770	1.70
	MediaTek				205	854	4.17
GUTIReallocationCommand (NAS)	Samsung	GUTIReallocationComplete	-	GUTIReallocationComplete	2218	4837	2.18
	MediaTek	-	-	-	205	921	4.49
IdentityRequest (NAS)	Samsung	IdentityResponse	-	IdentityResponse	2219	4796	2.16
	MediaTek	-	-	-	205	728	3.55
SecurityMode (NAS)	Samsung	-	-	-	2217	3426	1.55
	MediaTek	SecurityModeReject	-	SecurityModeReject	205	1278	6.23
ServiceReject (NAS)	Samsung	-	-	-	2218	13627	6.14
	MediaTek				205	728	3.55
TAUAccept (NAS)	Samsung	-	-	-	2218	17168	7.74
	MediaTek				205	1034	5.04
TAUReject (NAS)	Samsung	-	-	-	2219	3840	1.73
	MediaTek				205	728	3.55
ActivateDefaultEPSBearer (NAS)	Samsung	-	-	-	2217	3414	1.54
	MediaTek				205	708	3.45
BearerResourceAllocationReject (NAS)	Samsung	-	-	-	2218	3409	1.54
	MediaTek				205	708	3.45
BearerResourceModificationReject (NAS)	Samsung	-	-	-	2218	3409	1.54
	MediaTek				205	708	3.45
DeactivateEPSBearerContextRequest (NAS)	Samsung	DeactivateEPSBearerContextAccept	-	DeactivateEPSBearerContextAccept	2217	6102	2.75
	MediaTek				205	2934	14.31
ESMInformationRequest (NAS)	Samsung	ESMStatus	-	ESMStatus	2218	6041	2.72
	MediaTek	-	-	-	205	2749	13.41
ModifyEPSBearerContextRequest (NAS)	Samsung	-	-	-	2219	3409	1.54
	MediaTek				205	708	3.45
Notification (NAS)	Samsung	-	-	-	2218	3399	1.53
	MediaTek				205	708	3.45
ESMStatus (NAS)	Samsung	-	-	-	2218	3399	1.53
	MediaTek				205	708	3.45
PDNConnectivityReject (NAS)	Samsung	-	-	-	2218	3409	1.54
	MediaTek				205	708	3.45
PDNDisconnectReject (NAS)	Samsung	-	-	-	2218	3409	1.54
	MediaTek				205	708	3.45
Average	Samsung				2281	5458	2.41
	MediaTek				203	1137	5.54

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

Existing techniques for detecting bugs in the baseband firmware are either too shallow (fuzzing) or too time-consuming (Over the air). To bridge this gap, this paper presents BaseBridge, an approach to emulate baseband behavior to detect bugs. Two key advantages of Basebridge is: 1) State extraction and 2) Dynamic memory restoring. BaseBridge transfers the state of the physical devices that are already connected to a cellular network and continues emulation from that state. This approach is used to fuzz the LTE protocol stack for the MediaTek and Samsung devices. Results presented in the paper show significant improvement in coverage rates achieved on the targets.

C.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability

C.3. Reasons for Acceptance

1) BaseBridge advances the state of the art in baseband emulation while reducing the need for manual intervention.
2) Achieving more coverage and consequently triggering new vulnerabilities