BASESPEC: Comparative Analysis of Baseband Software and Cellular Specifications for L3 Protocols

Eunsoo Kim*, Dongkwan Kim*, CheolJun Park, Insu Yun, Yongdae Kim
KAIST
{*hahah, dkay, fermioncj, insuyun, yongdaek*}@kaist.ac.kr

***Abstract*—Cellular basebands play a crucial role in mobile communication. However, it is significantly challenging to assess their security for several reasons. Manual analysis is inevitable because of the obscurity and complexity of baseband firmware; however, such analysis requires repetitive efforts to cover diverse models or versions. Automating the analysis is also non-trivial because the firmware is significantly large and contains numerous functions associated with complex cellular protocols. Therefore, existing approaches on baseband analysis are limited to only a couple of models or versions within a single vendor.**

In this paper, we propose a novel approach named BASESPEC, which performs a comparative analysis of baseband software and cellular specifications. By leveraging the standardized message structures in the specification, BASESPEC inspects the message structures implemented in the baseband software systematically. It requires a manual yet one-time analysis effort to determine how the message structures are embedded in target firmware. Then, BASESPEC compares the extracted message structures with those in the specification syntactically and semantically, and finally, it reports mismatches. These mismatches indicate the developer's mistakes, which break the compliance of the baseband with the specification, or they imply potential vulnerabilities. We evaluated BASESPEC with 18 baseband firmware images of 9 models from one of the top three vendors and found hundreds of mismatches. By analyzing these mismatches, we discovered 9 erroneous cases: 5 functional errors and 4 memory-related vulnerabilities. Notably, two of these are *critical* remote code execution 0-days. Moreover, we applied BASESPEC to 3 models from another vendor, and BASESPEC found multiple mismatches, two of which led us to discover a buffer overflow bug.

I. INTRODUCTION

A baseband processor (BP) of cellular devices such as smartphones plays an important role in cellular networks. Although users mainly interact with the interface of user applications running on an application processor, all application data are transferred by the BP to use its radio interface. The BP runs the software, which is typically a real-time operating system dedicated to managing the radio communication; therefore, it includes low-level digital signal processing and complicated cellular protocol stacks. To provide seamless network services to users, the baseband software continuously communicates with a core network using numerous cellular control plane messages at layer 3 (L3) [4].

The baseband software is an alluring attack target because it can be used to monitor and modify the transferring data if it is exploited. Therefore, researchers have proposed several approaches to analyze its security, particularly for the L3 protocols. To discover security bugs in their implementation, researchers dynamically analyzed specific protocols, such as SMS or cell broadcast messages, using fuzzers [47], [46], [63], [42], or they manually inspected a small portion of baseband software in an ad-hoc manner [25], [15], [64].

These approaches suffer from mainly three technical challenges: the obscurity of baseband firmware, limited applicability of manual analysis, and difficulties in automation. First, the structure of baseband firmware is obscure as vendors are reluctant to publish their details. Second, to uncover this obscurity, a manual analysis is inevitable, which requires significant repetitive efforts to investigate numerous functions (*i.e.*, over 90K) across diverse baseband models or versions. Third, automating is thus necessary, but it is also non-trivial. The size of the baseband software is extremely large (*i.e.*, tens of MB) and cellular protocols contain numerous complex states, which can be neither statically analyzed nor dynamically triggered by fuzzers. In addition, identifying bugs requires an explicit oracle, such as a program crash or a noticeable abnormal behavior; therefore, it is limited to a few bug types. Thus, existing approaches can analyze only a couple of device models or versions within a single vendor.

To address these challenges, we propose a novel system named BASESPEC, which performs a comparative analysis of baseband implementation and cellular specifications, by leveraging the nature of a baseband as a modem for network communication. BASESPEC's key intuition is that a message decoder in baseband software embeds protocol specifications in a machine-friendly structure to parse incoming messages; hence, the embedded protocol structures can be easily extracted and compared with reference to the specification. This enables BASESPEC to automate the entire comparison process and explicitly discover mismatches in the protocol implementation, which are non-compliant to the specification. These mismatches can directly pinpoint developers' mistakes when embedding the protocol structures or hint at potential vulnerabilities.

For the comparative analysis, BASESPEC first analyzes baseband firmware to identify the message decoder and extracts embedded protocol structures in the firmware. Then, it compares the extracted structures with the specification in two aspects: 1) whether the embedded structures are *syntactically* equivalent to the specification, and 2) whether the decoder function

*These two authors equally contributed.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24365
www.ndss-symposium.org

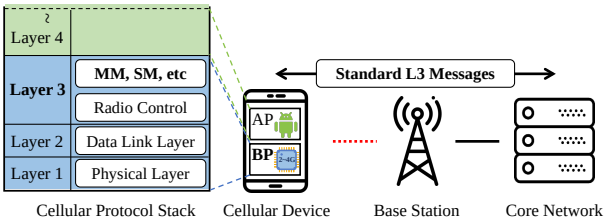

Fig. 1: Overall cellular network architecture

semantically follows the specification. Thus, a comparative analysis between the actual implementation and specification can explicitly identify mismatches. Then, we manually analyze those mismatches to verify if they can produce functional or security bugs. Note that BASESPEC requires an initial analysis of target firmware to locate the decoder function and determine how the message structures are embedded in the firmware. This may require considerable manual efforts; however, this analysis is only a one-time task. The knowledge obtained from this analysis can be reused for other baseband models or versions. This is because the main decoding logic rarely changes across diverse baseband models or versions within a vendor. Meanwhile, the automated comparison procedure can be reused across other vendors once their firmware is analyzed.

With a prototype of BASESPEC, we analyzed the implementation of standard L3 messages in 18 baseband firmware images of 9 device models from one of the top three vendors. BASESPEC identified hundreds of mismatches that indicate both functional errors and potentially vulnerable points in the baseband implementation. We investigated their functional and security implications and discovered 9 erroneous cases affecting 33 distinct messages: 5 of these cases are functional errors and 4 of them are memory-related vulnerabilities. Notably, 2 of the vulnerabilities are *critical* remote code execution (RCE) *0-days*. To evaluate the applicability of BASESPEC, we also applied BASESPEC to 3 models from a different vendor in the top three. Through this analysis, BASESPEC identified multiple mismatches, two of which led us to discover a buffer overflow.

In summary, our contributions are as follows:

- We propose a novel approach named BASESPEC for discovering bugs in cellular baseband software. BASESPEC performs a comparative analysis of embedded specifications in baseband software and documented ones.
- We demonstrate the practicality of BASESPEC. By running an automated prototype of BASESPEC on 18 baseband firmware images of 9 models from one of the top three vendors, we identified hundreds of mismatches, which are non-compliant with the specification.
- By further analyzing the mismatches, we discovered 9 erroneous cases, of which 5 are functional errors and 4 are vulnerabilities including 2 RCE 0-days. We responsibly disclosed all findings to the vendor.
- Applying BASESPEC to 3 firmware images from another vendor identified multiple mismatches, two of which cause a buffer overflow bug.

II. BACKGROUND

A. Cellular Architecture

A cellular network has three main components, namely, a cellular device, base station, and core network, as shown in Figure 1. These components have different terms per cellular generation; here, we use generic terms for simplicity. For example, NodeB, eNodeB, and gNodeB represent the base stations for 3G, 4G, and 5G, respectively.

TABLE I: Protocols grouping standard L3 messages. The first column represents a 4-bit protocol discriminator (PD). The fifth denotes the document version that we used for comparison (§VII); the last column represents whether the protocol is implemented in the baseband binary.

PD	Description	Abbrev.	Spec No.	Version	Implemented in firmware
0	Group Call Control	GCC	44.068	-	-
1	Broadcast Call Control	BCC	44.069	-	-
2	EPS Session Management	ESM	24.301	v15.8	✓
3	Call control; call related SS	CS	24.008	v15.8	✓
4	GPRS Transparent Transport Protocol	GTTP	44.018	-	-
5	Mobility Management	MM	24.008	v15.8	✓
6	Radio Resources Management	RR	44.018	v15.5	✓
7	EPS Mobility Management	EMM	24.301	v15.8	✓
8	GPRS Mobility Management	GMM	24.008	v15.8	✓
9	Short Message Service	SMS	24.011	v15.3	✓
10	GPRS Session Management	SM	24.008	v15.8	✓
11	non-call related Supplementary Services	SS	24.080	v15.1	✓
12	Location Services	LCS	23.271	-	-
14	Reserved for extension	-	-	-	-
15	Tests procedures	-	36.509	-	✓

A cellular device refers to any device located at the edge of a cellular network, and it allows users to access cellular services; the most common device is a smartphone. A cellular device usually has two separate processors for performance: an application processor (AP), on which the mobile operating systems and user applications run, and a cellular BP, where radio/digital signal processing is performed.

A base station offers a wireless connection to cellular devices. It transmits messages, which are from the core network to a cellular device and vice versa, through the radio interface. Thus, it is responsible for managing radio resources to provide users better service quality. A core network provides core procedures such as mobility management and session management, which include crucial user identification and security services such as encryption and integrity checks.

The cellular protocol stack has multiple layers as the OSI model. The air interface of a cellular network is at layers 1 and 2 of the OSI model. Various core procedure messages are delivered on layer 3. To appropriately handle these layers, the baseband of a cellular device also implements the cellular protocol stack. Further, to provide backward compatibility for cell coverage and roaming, the latest 4G/5G cellular devices also support earlier 2G/3G cellular technologies.

B. Cellular Specifications and Standard Layer 3 Messages

The cellular specification is defined by an international working group called the 3rd Generation Partnership Project (3GPP), which unites seven telecommunications standard development organizations such as ETSI. There are over 100 specification documents, and most documents have hundreds of pages. Because of their enormous quantity and complexity, many mistakes have been observed in their implementations [47], [46], [63], [64], [25], [15], [42], [39], [23], [54], [51], [57].

Among the various protocols and messages in the specification, the standard layer 3 (L3) messages are used in complex core procedures, such as mobility management, session management, or even cryptographic operations to protect private information of users. Thus, multiple vulnerabilities have been found in their processing routines [64], [25], [15], [42]. The standard L3 messages are not only limited to a specific cellular

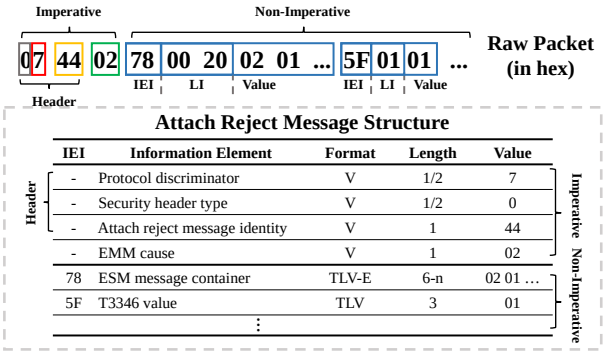

Fig. 2: EMM Attach Reject message structure

generation, such as GSM or LTE, but they also include several different protocols over generations [3]. Table I lists the L3 protocols, their protocol discriminators (PDs), and specification document numbers. Each specification document defines the detailed message formats and directions in the corresponding protocol. An L3 message can be transmitted to a cellular device (*i.e.*, downlink) or to the core network (*i.e.*, uplink). Furthermore, the L3 message may have different formats based on the direction in which it is transmitted. Hereinafter, we use the abbreviations listed in Table I to denote each L3 protocol.

Standard L3 message formats. Each standard L3 message has a specific format defined in the corresponding specification document (Table I) [3]. A standard L3 message starts with a 2-byte header that includes a PD and message identity. A tuple of a PD and message identity allows a recipient baseband to determine the format of a given message in order to decode the message. Each successive field of a message is referred to as a standard information element (IE).

A standard IE may have three parts: an IE identifier (IEI), a length indicator (LI), and a value, which are also referred to as a type (T), length (L), and value (V), respectively. An IE can be either *imperative* or *non-imperative*, according to the occurrence of an IEI. Imperative IEs do not have IEIs, whereas non-imperative IEs must have IEIs. In a message, imperative IEs must appear in a fixed order ahead of non-imperative IEs; thus, they can be distinguished without an IEI. An LI denotes the number of bytes for the value part, whereas the IE length in the specification represents the number of bytes for all parts. There are seven IE formats: `T`, `V`, `TV`, `LV`, `TLV`, `LV-E`, and `TLV-E`. Here, `T`, `L`, and `V` represent the occurrence of an IEI, LI, and value in an IE, respectively. The `-E` suffix extends the one-byte LI to a two-byte LI, which indicates a length from `0` to `65535`.

Figure 2 shows an example message called `ATTACH REJECT` in the EMM procedure with raw packet data. Each row in the message structure represents an IE. The header comprises the 4-bit PD (`0x7`), 4-bit security header type (`0x0`), and 8-bit message identity (`0x44`). Further, imperative IEs include a non-header part IE, `EMM cause`. The packet then continues with non-imperative IEs: `ESM message container` and `T3346 value`, of which IEIs are `0x78` and `0x5f`, respectively. Because the format of the `ESM message container` IE is `TLV-E`, its LI can indicate a length from `0` to `65535`. The length of the `T3346 value` IE is fixed to be `3` although it takes an LI.

C. Baseband Processor

In a cellular device, a BP is a dedicated processor responsible for managing all radio functions for cellular communication including digital signal processing. To meet the real-time requirements for radio communication, it runs a real-time operating system as its firmware Thus, its firmware operates as a single executable and is loaded into memory at runtime; thus, we refer to baseband firmware as a *baseband binary*.

Baseband software is typically proprietary, and manufacturers do not publicly share detailed information such as the source code. For instance, Qualcomm's Snapdragon, MediaTek's Helio, and Samsung's Exynos are the top 3 system-on-a-chip products that contains a BP [17]. However, none of these manufacturers share the detailed information. Therefore, researchers often perform reverse engineering to analyze and identify security problems in the baseband software. In addition, each baseband may have a different architecture according to its design choice. Thus, baseband analyses require an appropriate tool that supports the target baseband architecture.

L3 message processing. In the baseband, an L3 message is first classified by its PD and message identity. Then, the baseband parses and decodes the message to obtain the information of the IEs, using the pre-defined message structures. After decoding the message, it performs an appropriate action for each decoded IE, and finally, it processes the message as defined in the specification. Hereafter, we refer to the functions involved in the decoding procedure as *L3 message decoders*, and those for the message processing as *IE* and *message handlers*.

III. Overview

A. Challenges

There are mainly three technical challenges that hinder the existing approaches in finding the bugs in baseband firmware.

C1. Obscurity of baseband firmware. Cellular baseband firmware remains largely unknown because vendors do not make its details public to protect their proprietary implementations. This obscurity severely hinders the analysis of the firmware, thereby requiring significant manual efforts for the analysis. To reduce the manual efforts, memory dumps are often used as they already processed the initializing steps and include runtime information [25], [15]. However, obtaining them requires real devices, and a special feature (e.g., a hidden dump menu only available in old Android devices) or a vulnerability to trigger it. One can consider using a hardware debug interface, such as JTAG; however, it is also disabled in recent devices. Further, even memory dumps cannot be analyzed by state-of-the-art binary analysis tools such as IDA Pro [31]; for example, function identification, which is fundamental for a static analysis, often fails because of the firmware's obscurity.

C2. Limited applicability of manual analysis. To uncover the obscurity of baseband firmware, researchers have focused on a manual analysis [64], [25], [15]. However, this method is fundamentally limited in scalability and applicability because manually investigating numerous functions for hundreds of L3 messages is nearly impossible. Therefore, even similar types of vulnerabilities often remain undiscovered. In addition, as mobile devices are quickly evolving in their software and hardware, their firmware binaries have significant differences one another. Consequently, inspecting the firmware of diverse device models or versions, even within a single vendor, remains challenging, requiring additional, serious manual efforts.

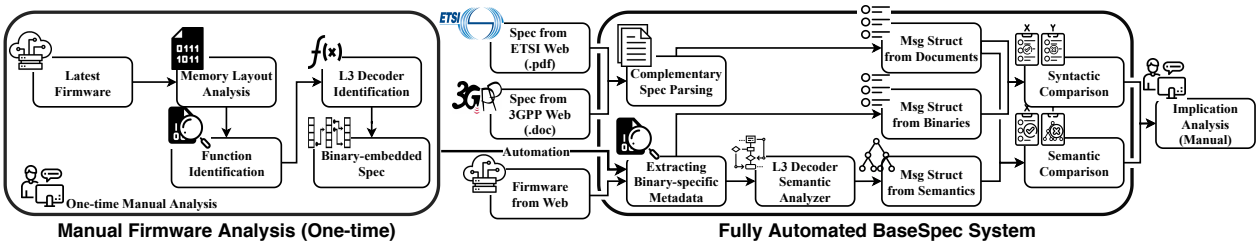

Fig. 3: Overview of our approach: manual firmware analysis and automated BASESPEC.

C3. Difficulties in automated analysis. Automating the baseband analysis is essential to achieve scalability and applicability. An automated analysis can be largely divided into static and dynamic analyses; however, both methods have several challenges. A static analysis suffers from that baseband firmware is extremely large (*i.e.*, tens of MBs), and it contains numerous non-trivial features to analyze, such as cryptographic operations. Moreover, building analysis rules is challenging because cellular specifications are quite complex, written in over 100 documents. Therefore, existing studies highly rely on a dynamic analysis (e.g., fuzzing) with real or emulated hardware [47], [46], [63], [64], [42]. Unfortunately, many vulnerabilities in the baseband are difficult to trigger dynamically owing to its convoluted states. These approaches also rely on an explicit oracle such as a program crash to identify bugs, thereby limiting them to a few bug types.

B. Our Approach

To tackle these challenges, we propose a novel approach named BASESPEC, which performs a comparative analysis of baseband firmware and cellular specifications. BASESPEC leverages the natural characteristics of a message decoder in network communication. Our key intuitions are that 1) a message decoder in the network communication needs to embed specifications in its implementation, particularly the message structures, to be able to identify and parse the message fields; 2) As such embedded message structures exist in a machine-friendly form, we can certainly extract them; 3) Comparative analysis on the extracted structures may identify incorrectly embedded ones with reference to the specification documents; 4) As the main logic of decoding routines rarely changes, message structures can be extracted across diverse device models/versions, similarly. Hereafter, we refer to the embedded specifications and message structures in baseband firmware as *binary-embedded specifications/messages*.

A workflow of our approach is illustrated in Figure 3. Our approach is largely divided into two parts: manual firmware analysis and fully automated BASESPEC. The firmware analysis mainly explores where the message decoder are located (§IV-C) and how specifications are embedded (§IV-D) in the baseband firmware, which we refer to as *binary-specific metadata*. This procedure is a manual, yet one-time task because the main logic of decoding procedure rarely changes across diverse device models or versions within the same vendor. Then, BASESPEC utilizes these results for syntactic/semantic comparison. Specifically, BASESPEC automates the extraction of the decoder function address and embedded message structures from the target baseband binary. Syntactic comparison literally validates whether the binary-embedded specifications match those of the documentations (§V-C). Meanwhile, semantic comparison investigates whether the underlying logic of the decoder function correctly follows the specification leveraging symbolic execution (§V-D). Finally, it reports mismatches, which indicate developers' mistakes, which may break the compliance with the specification, or imply potentially vulnerable points for later analysis. Thus, we only need to analyze the messages affected by the reported mismatches.

We address the challenges described in §III-A as follows. First, we uncover the firmware's obscurity by manually analyzing the firmware, particularly the L3 message decoders (*C1*). This analysis can be reused for other baseband firmware; *i.e.*, it is a one-time task. Second, BASESPEC automatically identifies mismatches from numerous L3 messages and reveals potentially buggy points for analysis (§VII-B); thus, it enables an efficient and practical analysis (*C2*). Indeed, by analyzing the mismatches reported by BASESPEC, we discovered 9 error cases, of which 5 are functional errors and 4 are vulnerabilities including 2 RCE 0-days (§VII-C). Finally, as the main decoding logic rarely changes, BASESPEC is applicable to various device models or versions with automation (*C3*) (§VII-D). It can be also applied to other vendor's firmware although it requires one-time manual efforts to analyze its decoder (§VII-E).

C. Scope of This Work

Among the various protocols in the cellular network, we choose standard L3 messages as our target. These messages include various protocols and play an important role in the cellular core procedures (§II-B). As the L3 protocol has numerous complicated logic and data structures, several vulnerabilities have been discovered in their implementations [25], [15], [42]. Therefore, we focus on analyzing the standard L3 messages as listed in Table I. Not all messages in L3 protocols are marked as the standard L3 messages; these other messages are beyond the scope of this study (§IX). Please refer to §II-B for exact definition of the standard L3 message.

For the cellular baseband, we mainly focus on one of the top three mobile processor vendors [17] (*i.e.*, $\texttt{Vendor}_1$). We analyze their baseband firmware on multiple latest device models (Table IV) whose architecture is ARM.[1] Our approach can apply to other baseband firmware; however, it may require considerable manual efforts to uncover the obscurity of their firmware. We additionally analyzed the firmware of another one of the top three vendors (*i.e.*, $\texttt{Vendor}_2$) and successfully applied BASESPEC (§VII-E).

IV. MANUALLY UNCOVERING FIRMWARE OBSCURITY

This section details our approach to uncover the obscurity of baseband firmware. We describe how we handle several issues in the state-of-the-art static analysis tool named IDA Pro (§IV-B),

[1] We anonymized the names of devices and vendor upon the vendor's request.

locate the L3 decoding function (§IV-C), and determine the way of message structures are embedded (§IV-D). Recall that the obscurity and complexity of baseband firmware makes a manual analysis essential (§III-A). However, the analysis procedure is a one-time task, of which results can be reused for multiple messages, models, or versions within the same vendor; this will be shown in §VII-D. This section mainly shares our experience in analyzing $\texttt{Vendor}_1$'s firmware. However, similar approaches can be applicable to other vendors' firmware, although it may require considerable efforts to determine their L3 message decoding logic, and we will describe this in §VII-E.

A. Firmware Acquisition

We chose baseband firmware as our analysis target without requiring physical devices because of its applicability and accessibility. There are mainly two methods to obtain baseband firmware: memory dumps and firmware images. Previous studies [25], [15] relied on a memory dump because it contains runtime memory states, a memory layout, and global variables, which do not require complicated analyses for firmware initialization. However, this method requires a real device to dump memory; hence, it significantly degrades scalability and applicability. Moreover, we found that the hidden menu to trigger the memory dump or a hardware debug interface, such as JTAG, has been disabled in the recent devices.

Therefore, we decided to use a third-party website [62] that maintains firmware images for updates. One can also download the latest firmware images from the vendor's cloud storage. However, the third party storage provides a well-structured list of firmware images per product model and version. Among them, we selected the images of the latest flagship models as listed in Table II. For an initial firmware analysis, we need to analyze only a single image to break down its obscurity. Then, we can apply the knowledge to other images within the same vendor. Therefore, we selected to analyze the latest version of the latest model (*i.e.*, *Model A* in Table II).

B. Preprocessing

We analyze the baseband firmware with IDA Pro [31], which is a state-of-the-art binary analysis tool. Although IDA Pro is a promising tool, it requires additional preprocessing steps for 1) memory layout analysis and 2) function identification. Because of several run-time mechanisms in the baseband firmware, IDA Pro correctly identifies only hundreds among 90K functions in it. Such a limitation is already known to be challenging [9]. Without any preprocessing, IDA's automatic analysis can detect only 450 functions starting from interrupt handlers, which are common entry points of embedded devices (see Table II). Therefore, we design the two preprocessing steps as follows.

Memory layout analysis. For analysis, the baseband firmware should be loaded in a proper memory layout. Otherwise, data or function pointers in the firmware would point to invalid memory addresses, which significantly hinders further analysis. Indeed, when we opened the firmware using IDA, the data/function pointers in most functions attempted to access data or call other functions located in invalid memory addresses.

We discovered that this invalid pointer issue is caused by *scatter-loading*, and IDA fails to support it. Scatter-loading is an ARM's loading mechanism that reallocates the initially loaded file to multiple memory regions at runtime. This technique is widely used in ARM-based embedded systems because it supports compression of data regions, thereby reducing the firmware size. When building firmware, a component in the ARM compiler, named `armlink`, inserts functions for scatter-loading for initializing the firmware at runtime. These functions copy, decompress, or zero-initialize the memory regions according to a predefined table to properly set up the memory layout. Therefore, without handling the scatter-loading, an entire binary file is loaded into a single continuous memory region, which makes the data/function pointers invalid.

To handle the scatter-loading issue and create a proper memory layout, we emulate the scatter-loading process. Specifically, we mimic the behavior of the scatter-loading functions: we copy, decompress, and zero-initialize the memory regions. Because those scatter-loading functions are predefined by `armlink` in highly optimized forms, most recent ARM embedded devices reuse these functions. Thus, we can detect these functions with signatures similarly to IDA's FLIRT [30]. After detecting the scatter-loading functions, we analyze their cross-references and identify the predefined scatter-loading table. This table contains information that indicates the execution sequence and parameters of the scatter-loading functions. We emulate the scatter-loading process as stated in the table.

Function identification. Our target (*i.e.*, $\texttt{Vendor}_1$'s firmware) is based on the ARM architecture. To identify functions in the firmware, we need to disassemble its byte code in advance. However, disassembling unknown bytes in ARM is error-prone [36] because the ARM architecture supports two instruction sets: ARM and Thumb. The ARM instruction set is the default mode that executes 32-bit instructions, and the Thumb instruction set supports compact 16-bit instructions to reduce the code size. Because the same bytes can be disassembled in two different instructions, direct disassembling would lead to many incorrectly disassembled codes.

To tackle this challenge, we designed two simple techniques that leverage i) frequent function prologues and ii) the characteristics of function pointers in the Thumb mode. First, we build function prologue signatures that can distinguish between the ARM and Thumb modes by investigating the identified functions. These prologue signatures comprise `PUSH` instructions in both ARM and Thumb mode. We then search those signatures; if a match is found, we attempt to analyze it in the mode of the matched signature. To reduce false positives in signature-based matching, we verify whether the function prologue handles registers normally. For instance, most functions push the `LR` register in stack but do not push the `PC` register; hence, we discard prologues that push the `PC` register or those that do not push the `LR` register.

After detecting the function prologues, we further identify functions by analyzing function pointers in the data section. For this, we leverage the characteristics of the Thumb mode. Function pointers to the Thumb mode functions use odd numbered addresses; particularly, the least significant bit of the pointer value is always `1`. As most data are aligned with an even address, an odd numbered address that points to a code section can be a Thumb mode function pointer. Therefore, we can find functions that are called indirectly via such pointers.

Preprocessing results. These preprocessing techniques significantly improve the IDA's performance in identifying functions.

TABLE II: Number of newly identified functions in baseband firmware by our preprocessing; numerous functions were hidden under the IDA Pro's default analysis (i.e., before our preprocessing).

Model	Firmware Build Date	# of Funcs: Firmware Size (MB)	# of Funcs: Before Our Preprocessing	# of Funcs: After Our Preprocessing	Elapsed Time (s)
Model A	May/2020	44.09	450	91,481	8,442
Model B	May/2020	44.06	3,505	90,519	2,219
Model C	May/2020	43.82	444	90,328	3,171
Model D	Jun/2020	41.38	409	73,199	1,451
Model E	Jun/2020	41.40	409	83,725	3,434
Model F	Apr/2020	41.78	410	39,003	1,045
Model G	Apr/2020	41.21	410	31,888	1,112
Model H	Apr/2020	37.46	380	35,216	790
Model I	Apr/2020	37.09	379	58,974	1,347

The initial number of functions identified by IDA Pro was 450 for *Model A*, as shown in Table II. We applied the memory layout analysis and function identification, which consists of the prologue detection and pointer analysis, to the same firmware. Our memory layout analysis found 504 new functions, and our function identification techniques, i.e., the function prologue detection and function pointer analysis, found 31,955 and 2,526 new functions, respectively. If we give these newly identified functions to IDA Pro, it further analyzes the code references in each function and recursively finds more functions. Consequently, our preprocessing steps helped IDA Pro to identify 91,481 functions eventually. The preprocessing is merely a one-time task, and it can apply to the firmware of other device models without any other manual effort. In practice, we can successfully preprocess the other latest models as listed in Table II. The average time spent on the full preprocessing including the IDA's auto-analysis was 2,557 s. Sometimes, IDA found more functions before the preprocessing (*Model B*'s case), or its auto-analysis required more time (*Model A*'s case). We are inspecting these outliers.

C. Identifying Layer 3 Decoder

To investigate the standard L3 messages (§III-C), we first need to locate their decoding logic through binary analysis. We call the functions implementing this decoding logic as *decoder functions*. We focus on the decoder functions because they have machine-friendly information for the L3 messages structures. As described in the §II-B, the L3 protocol messages have a standardized structure. To properly parsing the messages, developers embed message structures from the specification, in a machine-friendly format. Therefore, we can analyze the embedded structure systematically and understand how the firmware decodes the L3 messages.

To identify the decoders, we utilize debug information (e.g., logging messages) left by developers in the baseband binary. Note that this debug information is different from that inserted by compilers with the `-g` option, which disappear if a binary is stripped. We describe the details in the next paragraph. We select this approach among various methods in binary analysis because debug information is commonly used in practice for finding a specific function in a stripped binary when analyzing embedded devices [20], [25], [15]. Baseband firmware is stripped and extremely large (over 30MB) composed of numerous functions (over 90K), which makes it significantly challenging to discover an L3 decoder without such information. Therefore, we utilize debug messages and share their details below as an example for further research. Similarly, we found an L3 decoder function in another vendor's firmware using debug

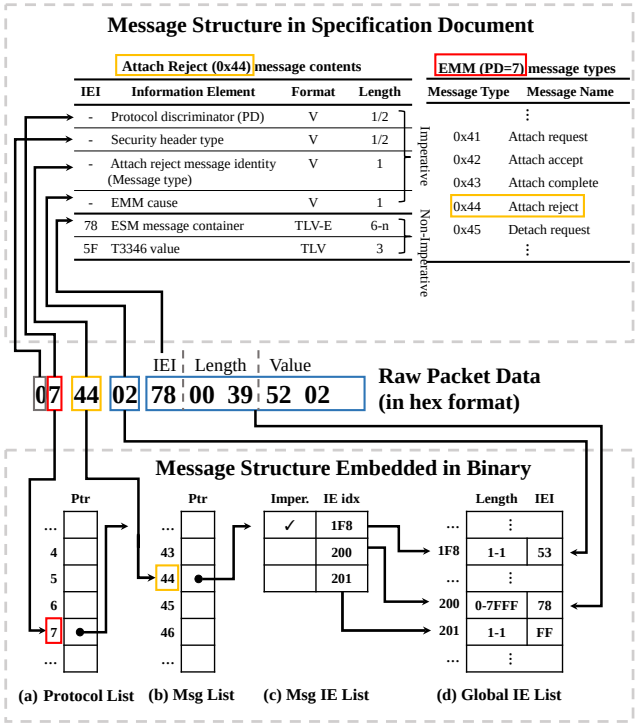

Fig. 4: Relationship between specification document and binary

information, although the structure of debug information in it is different (§VII-E). Meanwhile, our ultimate goal, which is conducting a comparative analysis on the standard L3 messages, does not depend on how the decoder function is identified. Other techniques can also be used for this process.

We first search all debug information in the baseband binary and then analyze functions that refer to only the corresponding debug information of interest. While searching debug information, we noticed that the firmware uses a specific structure to log debug messages and information. The structure, which starts with a magic value `DBT`, contains a debug message along with the file path and the line number where it is referenced. Therefore, we first search all debug information using `DBT`. Because numerous functions reference the debug information indirectly (≈100K cases), we perform a lightweight backward slice analysis to match the debug information with functions correctly. Next, we categorize each function based on the file path in the debug information, as the functions in the same layer or library may share the path. After categorization, we find L3 functions using debug messages and paths that contain keywords such as `L3`, `SS`, `EMM`, or `NAS`. Then, we find functions related to decoding incoming messages using keywords such as `decode`, `codec`, and names of several IEs. Consequently, we identified a function that parses standard L3 messages. We found that a single decoder function decodes all standard L3 messages regardless of their protocols, as these messages have the same standardized structure (§II-B). Therefore, a single decoder would suffice to handle them.

D. Obtaining Binary-embedded Message Structures

Finally, we determine how the standard L3 message structures are embedded in the baseband binary. To achieve this, we analyze the decoder function and its data references. The simplified architecture of the embedded message structure is illustrated in Figure 4 with the `EMM ATTACH REJECT` message

as an example. The embedded message structures are encoded as a hierarchical structure of four types of lists:

- `Protocol List` (Figure 4 (a)) is the top-level list of the hierarchy. It holds pointers to the `Msg List` of each L3 protocol and is indexed by PDs. As the PD of the EMM protocol is `7`, the 7th item in the `Protocol List` is accessed in the example.
- `Msg List` (Figure 4 (b)) is defined for each protocol. It holds pointers to the `Msg IE List` of each message in the protocol and is indexed by message identity values. In the example, as the message identity of the `EMM ATTACH REJECT` message is `0x44`, its `Msg IE List` is accessed using `0x44`.
- `Msg IE List` (Figure 4 (c)) is defined for each message. It contains the imperative flag and index for each IE in the message. The imperative flag shows whether the IE is encoded as imperative or non-imperative in the message, and the index represents the location of the IE in the `Global IE List`. As shown in the example, the first three of the IEs — PD, security header type, and message type — in the `EMM ATTACH REJECT` message are not listed in the `Msg IE List` because they are common IEs for all messages.
- `Global IE List` (Figure 4 (d)) contains information of all IEs used in the L3 protocols and is accessed by the index assigned for each IE. The information consists of the length and IEI of the IE. Note that the length here indicates the size of only the value part, whereas the length in the specification document indicates thee total size of the IE (§II-B).

We extract all embedded message structures by iterating these lists. From the firmware analysis, we can eventually notice how to obtain the L3 decoder address and message structure information in the firmware. With this knowledge, we automate BASESPEC as described in the next section.

V. BASESPEC DESIGN

This section describes BASESPEC's design in detail. Figure 3 illustrates the overview of BASESPEC. BASESPEC automatically reports mismatches by comparing the message structures in the baseband firmware and those in the specification documents. To this end, BASESPEC first extracts messages structures from specifications (§V-A) and those from the firmware with the binary-specific metadata (§V-B). Then, it syntactically compares the structures in the specification to binary-embedded ones (§V-C) and semantically examines their implementation logic using symbolic execution (§V-D). Based on these comparisons, BASESPEC reports various types of mismatches between the specification and implementations. BASESPEC reports suspicious IEs that exist only in binary (i.e., unknown mismatches) or specifications (i.e., missing mismatches). Moreover, it reports IEs with different length (i.e., invalid mismatches). After obtaining the mismatch results, we can further analyze their implications (§V-E).

A. Extracting Message Structures from Specification Documents

To inspect L3 message structures in the baseband firmware, BASESPEC extracts reference structures from the specification documents. The 3GPP and its partner organizations provide the specification documents on their websites [1]. BASESPEC downloads the latest specification documents listed in Table I and converts them into a raw text format. Then, it extracts the message structures from the converted raw text using regular expressions. The message structures in the specification documents include two parts, namely, *message contents*, which are a list of IE formats, and a list of *message types* for each L3 protocol, as illustrated in Figure 4. BASESPEC automatically parses those structures for each standard L3 message.

Although this text processing seems trivial, BASESPEC needs to address several problematic situations listed below.

Conversion errors. Converting a specification document into a raw text format introduces several types of errors. The specification documents are in human-friendly forms such as Microsoft Word (*i.e.*, `DOC`) or Adobe (*i.e.*, `PDF`) format. Their visual richness (e.g., tables and figures) helps readers understand these documents more thoroughly. However, BASESPEC needs to convert the documents into machine-understandable formats for a systematic analysis; converting these human-friendly documents into the raw text format relies on error-prone methods such as OCR [24]. Thus, such conversion often results in several errors including incorrect or missing words/sentences.

To mitigate such conversion errors, BASESPEC co-utilizes different document formats. The 3GPP and ETSI provide the same specification documents in two different formats: `DOC` file on the 3GPP and `PDF` file on ETSI. We found that conversion errors from each format are deterministic and complimentary. For example, when processing the specification documents of EMM and ESM messages (Table I), the conversion of tables for message types in the `DOC` files failed, whereas the conversion of `PDF` files was successful. In contrast, converting the tables for message contents showed the opposite case. Therefore, BASESPEC selects the correct raw text between different conversion results by checking the number of rows of converted tables; a table having more rows is more likely to be the correct one. Surprisingly, this approach produced no error. For conversion, BASESPEC utilizes `antiword` and `pdftotext` for `DOC` and `PDF` files, respectively.

Word inconsistencies. BASESPEC has to address many inconsistent words in the specification documents for text processing. As the specifications are manually written by numerous people, such inconsistencies are inevitable, which makes it difficult to parse the specifications systematically. These inconsistencies include 5 cases of duplicate and/or missing words, 14 cases of incorrect spaces between words, 5 cases of abbreviation usages, 14 cases of incorrect delimiters, and several different terms for denoting a single meaning. For example, `SYSTEM INFORMATION TYPE 15`, which is an RR message [7], is sometimes written as `SYSTEM INFORMATION 15`. In addition, there are four different names for denoting downlink (DL) messages which are transferred to a cellular device: UE, mobile station, MS, and DL. Furthermore, there is a missing delimiter `'-'` in the length of one IE format in the `DTM ASSIGNMENT COMMAND` message [7]. Some table names have duplicate words, such as `Contents of Service Request message content` [4]. We addressed all these inconsistencies and successfully retrieved message information for comparison. We reported the issues to 3GPP, so that they could be corrected for future research applying text processing in this field.

Irregular IE formats. While extracting the message structures from the specification documents, we found several nested IEs and invalid IE formats. For example, some SMS messages could have nested messages [5]; thus, the IEs of the nested

messages must be checked. We flattened the nested IEs to compare the message structures properly. Moreover, the `CN to MS transparent information` IE in the `INTER SYSTEM TO UTRAN HANDOVER COMMAND` message has an invalid `TLV` format as it does not have an IEI; an IE with the `TLV` format should include an IEI (§II-B). However, this was an exceptional case defined in the specification [7]. We made exceptions to handle the above cases when comparing the results.

B. Extracting Binary-specific Metadata

For further analysis, BASESPEC extracts binary-specific metadata: the information of binary-embedded message structures for syntactic comparison, and the address of the L3 decoder for semantic comparison. These are distinct across different baseband binaries. However, BASESPEC can extract this information regardless of the baseband binaries, and it is applicable for multiple baseband models or versions (§VII-D).

Given a firmware image, BASESPEC performs all firmware analysis procedures described in §IV and extracts the binary-specific metadata. For automating the firmware preprocessing (§IV-B), BASESPEC searches pre-built signatures of functions related to scatter-loading similarly to IDA Pro's FLIRT [30]. Then, it emulates their corresponding functionalities of copy, decompress, and zero-initialize. BASESPEC then scans the loaded firmware to detect function prologues and pointers for the Thumb mode functions. For automating the L3 decoder identification (§IV-C), we implement backward and forward slicers to identify L3-related debug structures correctly. Then, we can identify the L3 decoder by cross-referencing the debug structure. Finally, BASESPEC locates the address of message structures from the decoder as the function references the structures while decoding L3 messages. The message structures are used in the syntactic comparison, and the information regarding the decoder is used in the semantic comparison.

C. Syntactic Comparison of Message Structures

BASESPEC first *syntactically* compares message structures extracted from the baseband binary with those from the specification documents, at the IE-level granularity. For each message from the specification, BASESPEC fetches the corresponding message from the binary using PD and a message type (§V-C1). Next, BASESPEC iteratively maps IEs in the message from the specification onto those from the binary according to their types (i.e., imperative or non-imperative) (§V-C2). Finally, BASESPEC compares the mapped IEs and reports the mismatches, which we refer to as *syntactic mismatches* (§V-C3). These syntactic mismatches can directly identify developers' mistakes in embedding message structures in the baseband binary. We detail the syntactic comparison procedure as follows:

1) Fetching messages: For each message in the specification, BASESPEC first fetches its corresponding message structure from the baseband binary using the PD and message identity. As shown in Figure 5, BASESPEC fetches the corresponding `Msg IE List` for the `EMM ATTACH REJECT` message using a PD (`0x7`, red boxes) and a message identity (`0x44`, yellow boxes) as indices for the `Protocol List` and `Msg List`, respectively.

2) Mapping IEs: Next, BASESPEC maps each IE from the `Msg IE List` onto that in the specification. BASESPEC performs this mapping according to the IE type as an imperative IE and a non-imperative IE have distinct formats. For an imperative IE, BASESPEC relies on its order as it has a fixed order in the message. For example, in Figure 5, BASESPEC concludes that the first entry of the `Msg IE List` represents the `EMM cause` IE as it is the first IE, of which the imperative flag is set. Note that the `Msg IE List` only contains IEs after the message identity as the header IEs of the message (i.e., PD and the message identity) are already used to obtain `Msg IE List`. For a non-imperative IE, which can appear in an arbitrary order, BASESPEC uses its IEI, which is an identifier to distinguish it. For example, BASESPEC regards the second entry of the `Msg IE List` in Figure 5 as the `ESM message container` IE because its IEI (`0x78`) matches that in the specification.

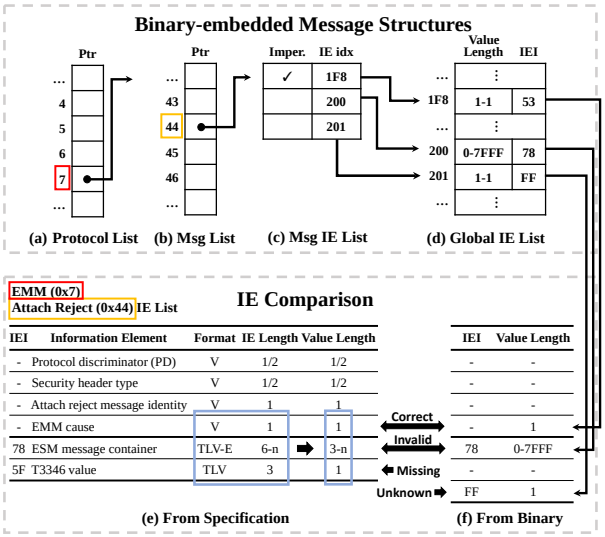

Fig. 5: Example of syntactic comparison

After the mapping process, BASESPEC reports the remaining IEs, which are not mapped, as either *missing* or *unknown* mismatches. *Missing* mismatches indicate IEs that exist in the specification but are not implemented in the binary. Meanwhile, *unknown* mismatches refer to IEs that exist only in the binary. For example, BASESPEC fails to map the `T3346 value` in Figure 5 as its IEI (`0x5F`) does not exist in the `Msg IE List`. Therefore, BASESPEC reports this as a missing mismatch. Similarly, BASESPEC reports the third IE in the `Msg IE List` as an unknown mismatch because its IEI (`0xff`) has no corresponding IE in the specification.

3) Comparing IEs: BASESPEC finally compares IE pairs from the mapping and reports the mismatch results. BASESPEC first needs to convert the IEs in the specification to a comparable format for the binary. Specifically, BASESPEC adjusts the IE lengths in the specification because lengths in the binary and specification are different. A length in the binary only considers the value part of an IE (*i.e.*, a `value length`), whereas that in the specification also includes IEI and LI (*i.e.*, an `IE length`). BASESPEC subtracts the size of IEI and LI according to the format in the specification (§II-B). For example, as shown in Figure 5, BASESPEC subtracts three bytes from the `IE length` of the `ESM message container` IE to calculate its `value length` because its format includes a 1-byte IEI (`T`) and 2-byte extended LI (`L` with `-E`). Similarly, BASESPEC subtracts two bytes to the `IE length` of the `T3346 value` IE, which has an IEI (`T`) with a 1-byte LI (`L`). BASESPEC does not adjust the `IE length` of the `EMM cause` IE as it only has the value (`V`).

Then, BASESPEC compares the adjusted IEs. If their lengths are different, BASESPEC reports them as *invalid* mismatches. For example, in Figure 5, the `value lengths` of `EMM cause` IE is the same in both the specification and the binary; thus, BASESPEC does not report any mismatch. Meanwhile, the minimum `value length` of the `ESM message container` IE in the specification (`3` bytes) differs from that in the binary (`0` byte); thus, BASESPEC marks this as an *invalid* mismatch.

D. Semantic Comparison of Message Structures

In addition to syntactic analysis, BASESPEC performs *semantic* analysis. Although syntactic analysis can identify evident mismatches of the message structures, the actual decoding logic for a given message in the baseband binary could be different from its syntactic form. To this end, semantic analysis focuses on how incoming messages are parsed in the decoder function. BASESPEC reveals the semantic flaws of the decoder function by discovering mismatches in the handling of messages between the implementation and the specification; we refer to these mismatches as *semantic mismatches*. These semantic mismatches can imply unintended behavior of the baseband different from the specification.

For semantic analysis, BASESPEC symbolically executes the decoder function (§V-D1), whose address is given from §V-B. Then, BASESPEC converts constraints, which are obtained from symbolic execution, into IEIs and LIs using their distinct roles (§V-D2); an IEI distinguishes the non-imperative IEs, while an LI specifies the size of value part. Next, BASESPEC builds message structures based on the identified IEIs and LIs, compares them with structures in specifications similarly to the syntactic comparison, and finally reports mismatches (§V-D4). Figure 6 depicts an overall procedure of our semantic analysis with a sample `EMM ATTACH REJECT` message.

1) Symbolic execution: BASESPEC analyzes the decoder function instead of the entire baseband binary following the concept of under-constrained symbolic execution [50]. Under-constrained symbolic execution analyzes individual functions directly without running the entire binary for scalability. Accordingly, BASESPEC performs symbolic execution from the entry of the decoder function until it returns. For an efficient analysis, BASESPEC concretizes the PD and message type, so that it processes one L3 message at a time. The message body, namely IEs, remains unconstrained to consider any possible IEs. For example, the message in Figure 6 has concrete values for the PD (`0x7`) and message type (`0x44`), which indicate the `EMM ATTACH REJECT` message. However, the message body comprises of unconstrained symbolic variables (`v1`–`v4`).

Symbolic execution creates the symbolic variables and constraints; they contain the decoding semantics of IEI and LI. Each symbolic variable represents one of the IE fields (*i.e.*, IEI, LI, or value), and each constraint represents how the decoder processes the fields. In the decoder, conditional branches that are associated with the symbolic variables produce constraints of these variables. These constraints can be created by checking an IEI, in the case of non-imperative IEs, or verifying an LI, based on the embedded message structure. For instance, the program states in Figure 6 contain different constraints of symbolic variables depending on the paths they followed. The `S1` state, which includes a constraint of `v2==0x5F`, may have followed a path that decodes an IE with `0x5F` as the IEI value.

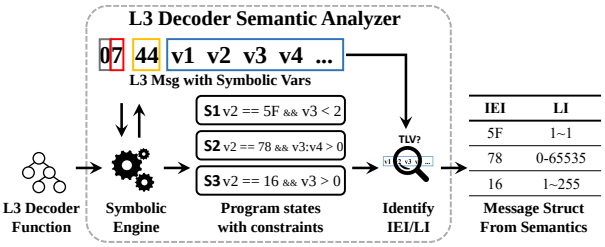

Fig. 6: Overview of semantic analysis

Meanwhile, `S2`, which includes a constraint of `v2==0x78`, may have followed another path comparing the IEI with `0x78`.

2) Identifying IEI and LI: When a symbolic state reaches the end of the decoder function, BASESPEC identifies IEIs and LIs from the collected symbolic variables and constraints. First, BASESPEC identifies an LI using its usages in memory addressing. As an LI specifies the size of the value part, the decoder function uses the LI in the address calculation to access the following IE. For instance, in Figure 6, suppose `v3` is an LI located at address `A`. Then, `v4` and its following bytes are the value part with the length `v3`. Thus, when the decoder function want to access the next IE, it will use `A+v3+1` as the address of the IE. Therefore, symbolic variables for an LI can be identified by checking whether they are used in any address. Further, a 2-byte LI that has a `-E` suffix format can be identified similarly. The last IE's LI cannot be identified in this manner; however, we can suppose the last unidentified symbolic variable as LI after other parts are identified.

Next, the IEIs of non-imperative IEs can be identified in a straightforward manner as they should be compared with predefined IEI values in the decoding routines. Thus, a symbolic variable is identified as an IEI if it is not an LI, and there are constraints that strictly confine its value. As shown in Figure 6, some constraints strictly limit the value of `v2` into `0x5F`, `0x78`, or `0x16`, which are possible values of the IEI. Therefore, `v2` is the IEI part. The value parts of IEs are identified implicitly as they are not constrained in the decoding routines. Symbolic variables may not even be accessed if the decoder function does not read the actual value and only stores the address of the value parts as an output. If the values are accessed and copied to other memory regions, we can identify such actions during symbolic execution and determine the value parts.

3) Handling of path explosion: In the process of symbolic execution, BASESPEC performs state pruning to prevent path explosion, which is a well-known problem from which symbolic execution-based approaches suffer [12], [13]. In particular, BASESPEC prunes a path that reached an error handling logic. If the decoder function detects obvious errors in the message, it invokes a complicated error handling logic. This error handling logic is irrelevant to legitimate message decoding; however, it causes path explosion. Therefore, we can discard this path and prevent path explosion. As the decoder function sets a flag variable to indicate its error, we can distinguish such paths with the flag variable. Thus, BASESPEC prunes paths which have set the flag variable.

In addition, we limit the number of non-imperative IEs analyzed in each state to prevent path explosion. Recall that non-imperative IEs can appear in an arbitrary order in a message. Therefore, numerous combinations of their sequences appear in symbolic execution, and this eventually produces numerous

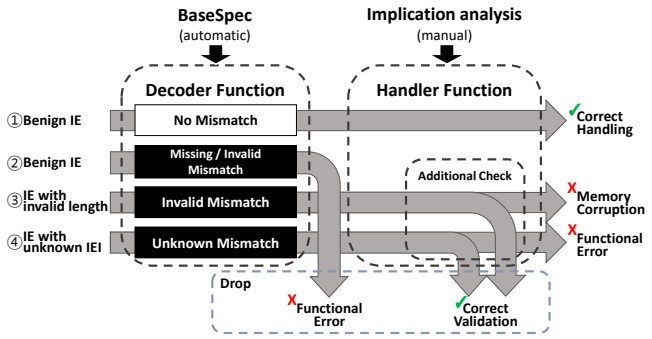

Fig. 7: Relationship between mismatches and their implications

states having complex constraints. To prevent state explosion, BASESPEC analyzes each non-imperative IE separately in an independent state, as most non-imperative IEs are optional and are not related to one another. Specifically, BASESPEC prunes a state if it has constraints of multiple non-imperative IEs, by periodically identifying IEIs in each active state. Note that all the imperative IEs are analyzed in each state because they must be present in a message.

4) Comparing IEs: For comparison, BASESPEC constructs semantic-aware message structures based on the identified IEI and LI values of the message. Each state from the symbolic execution has information of all imperative IEs and some non-imperative IEs. BASESPEC first composes the list of possible IEs by collecting the information from various states. A pair of an IEI and LI constructs a non-imperative IE, and an LI without an IEI builds an imperative IE. BASESPEC analyzes the semantics of the IEI and LI parts, and it does not identify imperative IEs without LI as those IEs have only the value part. For example, in Figure 6, the `S1` state constructs a non-imperative IE with `v2` as the IEI (`0x5F`) and `v3` as the LI. The `S2` state comprises a non-imperative IE with `v2` as the IEI (`0x78`) and `v3:v4` as the extended LI. Although the `ATTACH REJECT` message also has an imperative IE with `v1` in all states, it is not identified because it does not have an LI but only the value part. Then, BASESPEC constructs the message structure as the right table in Figure 6; it concretizes the LIs to show the explicit ranges of the lengths. The message structure is semantic-aware as it reflects the internal logic of the decoder.

Finally, BASESPEC compares the message structure with the specification documents similarly to that in syntactic comparison (§V-C). As imperative IEs must appear in a fixed order, BASESPEC compares their LIs sequentially, skipping imperative IEs without an LI. For non-imperative IEs, BASESPEC first matches them using their IEIs and then compares their LIs. BASESPEC reports any differences or remaining IE that are not matched as semantic mismatches.

E. Implication Analysis

Although BASESPEC automatically discovers mismatches between the specification and binary implementation, it requires additional manual analysis to understand impacts of the mismatches. When a message is given, the decoder function parses it and passes the message's IEs to the corresponding handler functions for further processing. BASESPEC automatically analyzes a decoding routine by leveraging a systematic structure of a message. However, a handler function has complicated semantics (e.g., session management or call control), which makes it difficult to validate its correctness automatically. Therefore, we rely on a manual analysis to analyze it. Nevertheless, it is worth noting that mismatches reported by BASESPEC can offer hints for this analysis; we need to analyze only the routines corresponding to the mismatched IEs rather than the entire function with complex logic.

TABLE III: BASESPEC's components and lines of code (LoC).

Component	LoC (Python)
Preprocessing	1,303 lines
Extracting binary-embedded specifications	2,105 lines
Parsing specification documents	566 lines
Syntactic comparison	546 lines
Semantic comparison	1,938 lines
Processing $\texttt{Vendor}_2$	749 lines
Total	7,207 lines

Figure 7 illustrates the relationship between mismatches in a decoder and the implications in handler functions. In particular, missing and unknown mismatches directly represent functional errors in the baseband firmware. As shown in ② of Figure 7, a missing mismatch causes a drop of benign IE; if an imperative IE (i.e., a mandatory field) is dropped due to the mismatch, it shows that the firmware fails to comply with the specification (i.e., functional errors). In addition, unknown mismatches are tightly coupled with missing mismatches. When a developer mistakenly embeds a wrong IEI value, both unknown and missing mismatches appear simultaneously. In such a case, the unknown mismatch directly represents a functional error.

Further, an invalid mismatch can have two implications; it can cause a functional error or memory corruption as it essentially represents that a decoder failed to validate the length of a certain IE properly. If the decoder's length limit for an IE is tighter than that defined in the specification, we do not need additional analysis because it represents a functional error that rejects a benign IE (② in Figure 7). Meanwhile, if the length limit is larger, it may cause memory corruption bugs in further processing (③ in Figure 7). For example, a buffer overflow can happen if developer blindly assumes a certain IE's length according to the specification although the actual length can be larger. As the handler function may have an additional check, it requires manual analysis on the handler to confirm the implications of invalid mismatches. Remarkably, such invalid mismatches provide us helpful insights on the developers' mistakes in embedding message structures in the baseband firmware, which lead us to discover several critical security vulnerabilities (§VII-C).

VI. IMPLEMENTATION

We implemented BASESPEC in 7k lines of code (LoC) in Python, as summarized in Table III. First, we utilized APIs in IDA Pro v7.4 [31] for automating the manual firmware analysis (§V-B). For the semantic analysis part of BASESPEC, we leveraged *angr*, a promising binary analysis framework [58], and we used its symbolic execution engine and constraint solver. To analyze memory access using length indicators, which are treated as symbolic variables (§V-D), we implemented the fully symbolic memory based on the approach of MemSight [19]. We released our source code that is irrelevant to the vendor to help further research.[2]

[2]https://github.com/SysSec-KAIST/BaseSpec

TABLE IV: Summary results of syntactic/semantic comparison and implication analysis. We anonymized the model names upon the request of the vendor; the names are assigned in an alphabetical ascending order from the latest one (i.e., Model A is the latest one). The mismatches show that the baseband binary can be non-compliant to the specification, but their implications have to be analyzed (see §V-E).

			In Binary		Common Mismatch					Syntactic-only Mismatch						Semantic-only Mismatch						Case Study Results								
			# of		Missing		Unknown		Invalid	Missing		Unknown		Invalid		Missing		Unknown		Invalid		Functional†					Memory-related			
	Model	Build Date	Msgs	IEs	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE	E1	E2	E3	E4	E5	E6‡	E7	E8‡	E9
Latest Firmware	Model A	May/2020	268	1204	1	164	0	36	38	109	3	19	6	13	21	52	1	6	0	9	35	203	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model B	May/2020	268	1201	1	167	0	36	38	109	3	19	6	13	21	52	1	6	0	9	35	200	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model C	May/2020	268	1201	1	167	0	36	38	109	3	19	6	13	21	52	1	6	0	9	35	200	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model D	Jun/2020	268	1200	1	179	0	36	41	111	3	18	6	13	21	52	1	6	0	9	32	186	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model E	Jun/2020	268	1200	1	179	0	36	41	111	3	18	6	13	21	52	1	6	0	9	32	186	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model F	Apr/2020	268	1198	1	179	0	36	41	111	3	18	6	13	21	52	1	6	0	9	32	186	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model G	Apr/2020	268	1198	1	179	0	36	41	111	3	18	6	13	21	52	1	6	0	9	32	186	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model H	Apr/2020	263	1096	1	212	0	3	40	39	4	19	8	34	21	118	1	327	0	1	32	71	·	✓	✓	·	·	·	·	✓	✓
	Model I	Apr/2020	263	1096	1	212	0	3	40	39	4	19	8	34	21	118	1	327	0	1	32	71	·	✓	✓	·	·	·	·	✓	✓
Oldest Firmware	Model A	Apr/2019	268	1216	1	170	0	36	38	109	3	19	6	13	21	52	1	6	0	9	35	197	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model B	Feb/2019	268	1213	1	173	0	36	38	109	3	19	6	13	21	52	1	6	0	9	35	194	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model C	Feb/2019	268	1213	1	173	0	36	38	109	3	19	6	13	21	52	1	6	0	9	35	194	✓	✓	✓	✓	✓	✓	·	✓	✓
	Model D	Mar/2018	269	1189	1	183	0	46	41	111	3	18	7	13	21	52	1	6	0	9	32	186	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model E	Mar/2018	269	1189	1	183	0	46	41	111	3	18	7	13	21	52	1	6	0	9	32	186	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model F	Apr/2017	269	1189	1	185	0	46	41	111	3	18	7	13	21	52	1	6	0	9	32	184	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model G	Apr/2017	269	1189	1	185	0	46	41	111	3	18	7	13	21	52	1	6	0	9	32	184	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Model H	Apr/2016	263	1096	1	212	0	3	40	39	4	19	8	34	21	118	1	327	0	1	32	71	·	✓	✓	·	·	·	✓	✓	✓
	Model I	Apr/2016	263	1096	1	212	0	3	40	39	4	19	8	34	21	118	1	327	0	1	32	71	·	✓	✓	·	·	·	✓	✓	✓

Common Mismatch: mismatches that are discovered by both comparisons in common. **{Syntactic, Semantic}-only Mismatch**: mismatches from only one comparison, **i-IE**: imperative IEs. **n-IE**: non-imperative IEs,
† Confirmed to be non-compliant to the Release 15 specification, ‡ 0-day vulnerabilities that lead to remote code execution (RCE).

VII. Evaluation

To evaluate BaseSpec, we answer the following questions:

- How effectively can BaseSpec discover specification mismatches? (§VII-B)
- How effective is the approach of BaseSpec in discovering new bugs of baseband? (§VII-C)
- How applicable is BaseSpec to inspect various firmware versions and device models? (§VII-D)
- How applicable is BaseSpec to firmware images from other vendors? (§VII-E)

A. Experimental Setup

Dataset. We collected firmware images from a third-party website [62], as described in §IV-A. We downloaded the latest and oldest firmware images from $\texttt{Vendor}_1$, which is one of the top 3 baseband chipset vendors. Note that the device models of these firmware images are officially supported by the vendor at the time of writing (July 1, 2020), as listed in Table IV. We chose the latest and oldest ones because they have the largest differences; thus, analyzing them can effectively show the applicability of BaseSpec on diverse versions/models (§VII-D). Further, we applied BaseSpec to another one of the top 3 vendors, $\texttt{Vendor}_2$ (§VII-E).

Specification. Among the various releases of the specification, we chose `Release 15` as its freeze point was on March 22, 2019. Therefore, the development for this specification was finished, and it was stable [2].[3] The detailed specification versions that we referred to are listed in Table I.

Machine. We ran all experiments on a server equipped with Intel Core i7-6700K at 4.00 GHz, 64 GB DDR4 RAM, and 2 TB SSD, operated on a Windows 10 operating system.

B. Comparative Analysis Results

Table IV shows the mismatches identified by BaseSpec. In summary, BaseSpec successfully discovered hundreds of mismatches in the firmware, which shows that its decoder does not comply with the specifications. In particular, BaseSpec extracts 277 standard L3 messages from the specification and compares them with messages implemented in the binary (the *# of msgs* column). For each message, BaseSpec checks three types of mismatches: *Missing*, *Unknown*, and *Invalid*. We count the number of mismatches separately for the imperative IEs (*i.e.*, *i-IE*) and for non-imperative IEs (*i.e.*, *n-IE*) because such IEs differentiate implications of mismatches as below.

Missing & Unknown IEs. BaseSpec found several missing IEs, which remain as unimplemented in the firmware binaries (*Missing* in Table IV). Theoretically, the baseband firmware should implement every IE in the specification to support complete functionalities defined in the specification. As non-imperative IEs are literally optional, they can be ignored in a message. However, imperative IEs must be present in a message; otherwise, the baseband cannot properly decode benign messages, thereby degrading the cellular service quality.

Further, BaseSpec discovered numerous unknown IEs that are not defined in the specification (*Unknown* in Table IV). The unknown IEs may be caused by incorrect implementations by developers. For example, if a developer incorrectly enters an IEI (*i.e.*, IE identifier), BaseSpec will consider the originally intended IE as missing, and the IE with the incorrect IEI as unknown. Therefore, these mismatches can locate functional errors that break compliance with the specification.

Invalid IEs. BaseSpec showed that hundreds of IEs are incorrect with the specification (the *Invalid* column). An IE consists of three parts: IEI, LI, and value. The decoder function uses only IEI and LI to validate a message and propagates the message to a message-specific handler. BaseSpec has already identified incorrect usages of IEIs as unknown mismatches, and it reports an IE with incorrect LI as invalid mismatches. Such invalid IEs imply that an improper message can be delivered to the handler function; therefore, if the handler fails to address this message properly, it can lead to a critical memory corruption vulnerability. For instance, the E6 bug, which will be described in §VIII, incorrectly assumes the maximum length of an IE, thereby resulting in buffer overflow for RCE.

[3] `Release 16` is just frozen at July 3rd, 2020.

Syntactic vs Semantic mismatches. BASESPEC employs syntactic and semantic methods for comparison, and both methods are complementary and essential. In particular, the syntactic comparison can find more missing/unknown IEs, while the semantic comparison can discover more invalid IEs (see the *Syntactic-only Mismatch* and *Semantic-only Mismatch* columns); the unusual number of mismatches in the *Model H* and *I* firmware will be described below. Both methods help us identify different error cases and their explicit causes. For example, mismatches in the syntactic comparison helped us pinpoint developers' mistakes regarding imperative IEs without LIs. Recall that such imperative IEs cannot be supported by semantic analysis (§V-D). Moreover, the numerous invalid IEs in the semantic comparison helped us discover the abnormal handling of IE lengths in the decoder function, which is inconsistent with the binary-embedded structures for syntactic comparison. This leads us to discover overflow vulnerabilities in handler functions. Consequently, these two methods are mutually complementary.

False positives. BASESPEC shows false positive results for two main reasons: limited support of the ARM architecture in angr and exceptional message structures in the baseband implementation. First, BASESPEC's semantic analysis could not fully process the *Model H* and *I* firmware, resulting in an unusual number of mismatches (see their results in the *Semantic-only Mismatch* column). We noticed that their binaries contain instructions that angr's symbolic execution engine, which is the basis of BASESPEC's semantic analysis (§VI), cannot completely support. In particular, most ARM instructions support conditional execution, which have a special suffix in their mnemonics such as `ADDEQ` and `CMPEQ`. However, angr currently does not support these instructions when the condition is symbolic. Thus, it raises errors at those instructions and fails to identify IEs correctly, resulting in missing IEs.

Second, we found that a few binary-embedded messages have exceptional structures. For instance, according to the specification, the `CC-ESTABLISHMENT` message only has the `Setup container` IE except for headers. However, BASESPEC reports multiple unknown mismatches as the message contains suspicious IEs that are not defined in the specification. After checking the specification, we found that the `Setup container` IE is a placeholder to contain the contents of a `SETUP` message, which consists of other multiple IEs. In addition, a message called `SECURITY MODE COMMAND` had four unknown imperative IEs. This message is designed to set up security parameters for encryption and integrity checks [6]. We found that the decoder handles this specific message exceptionally owing to its special purpose. Even though these mismatches are incorrect, we learned from these false positives that this message is special and worthwhile to be analyzed manually. As it requires more attention to implement such exceptional cases, they often involve critical vulnerabilities (§VIII).

C. Discovering Bugs with Mismatches

By investigating mismatches identified by BASESPEC, we successfully discovered 9 erroneous cases as shown in Table IV, which affect 33 distinct messages. We numbered those erroneous cases from E1 to E9. Five of them (E1–E5) are functional errors (the *Functional* column) that make baseband firmware non-compliant to the specification (e.g., rejecting a valid message), and the other four (E6–E9) are memory-related ones (the *Memory-related* column), which can lead to denial of service or even remote code execution. Except for E7, all of our cases are newly discovered (i.e., 0-days). We responsibly disclosed all of them to the manufacturer. In the following, we describe how BASESPEC helped us to discover these errors.

TABLE V: The number of missing mismatches for imperative IEs and unknown mismatches regarding each functional error in Model A.

	Common			Syntactic-only			Semantic-only		
	Missing	Unknown		Missing	Unknown		Missing	Unknown	
Errors	i-IE	i-IE	n-IE	i-IE	i-IE	n-IE	i-IE	i-IE	n-IE
E1	1	·	21	·	·	·	·	·	·
E2	·	·	2	·	·	·	·	·	·
E3	·	·	·	2	·	·	·	·	·
E4	·	·	·	·	2	13	·	·	7
E5	·	·	·	·	·	·	·	·	2
FP	·	·	13	1	4	·	·	·	·
Total	1	0	36	3	6	13	1	0	9

FP: false positives. **E1–E5**: functional errors

Functional errors from missing and unknown mismatches (E1–E5). As pointed out in §V-E, BASESPEC's mismatches are highly related to various types of bugs in baseband firmware. Missing imperative IEs and unknown IEs are strong indicators for functional errors. As listed in Table V, all such mismatches originated directly from function errors (E1–E5), except for false positives that were mentioned previously. Accordingly, we can identify such bugs from these mismatches. Note that one bug can cause multiple mismatches; for example, E1, which is the case for incorrectly ordered six IEs, causes a cascade effect on 22 mismatches, as shown in Table V. More interestingly, Table V summarizes the importance of employing both syntactic and semantic comparisons; we find E3 only from syntactic comparison and E5 only from semantic comparison owing to their individual advantages. Although these bugs do not have severe security implications, they may affect the service quality by disturbing the process of benign messages. Notably, E5 affects the `ATTACH ACCEPT` and `ROUTING AREA UPDATE` message, which are critical for network connection. Since how such functional errors affect cellular communication depends on the role of each communication protocol, verifying their effects is outside the scope of this study.

Memory corruptions from invalid mismatches (E6–E7). By checking invalid mismatches and related handlers, we discovered two memory corruption vulnerabilities (E6–E7). Unlike functional errors, memory corruption vulnerabilities require more manual efforts to understand security issues in handler functions; invalid mismatches can be harmless in terms of security because the handlers may have additional checks. However, invalid mismatches can help discover bugs because we can focus on the effects of the mismatches (i.e., non-compliance with specification) instead of analyzing every handler logic, which is extremely complex [25], [15], [64]. For example, we discovered E6 by focusing on one IE's length, which can be much larger than that in the specification; its length is 5 bytes in the specification, but the firmware allows it up to 255 bytes.

Other memory corruptions from failures (E8–E9). By analyzing two failure cases of BASESPEC, we discovered two more memory corruption vulnerabilities (E8–E9). BASESPEC produced false positives for a few messages as described in §VII-B, and it stopped running its semantic analysis for one message. BASESPEC analyzes a universal decoding routine in baseband firmware to check their compliance with the

specification. Therefore, the failure of BASESPEC for a certain message represents that this message is handled specially with its dedicated routine, which is error-prone. Consequently, we further analyzed the failures and discovered two memory corruption vulnerabilities. In particular, we discovered E8, which could be exploited for RCE, while investigating the false positives listed in Table V; they have exceptional structures unlike other messages. Moreover, we discovered E9 by analyzing a failure in symbolic execution for the `START DTMF ACKNOWLEDGE` message. We found that our symbolic execution engine (i.e., angr) reports a memory access violation error for that specific message unlike others; its report is indeed a vulnerability that dereferences an improperly initialized pointer variable, thereby causing a crash. Note that this vulnerability is detectable only with our efforts to address path explosion and implement symbolic memory.

D. Applying BASESPEC to Various Firmware Images

To check the applicability and scalability of BASESPEC, we ran BASESPEC to analyze all the collected firmware images from $\texttt{Vendor}_1$. As a result, we found that BASESPEC can effectively identify mismatches in all tested firmware images, as summarized in Table IV. The build dates of the latest images are in a two-month period, whereas those of the oldest ones are spread over four years. The average time spent on analyzing bare-mental firmware was 3,156 s, of which 2,557 s ($\approx 81\%$) were spent for preprocessing (§IV-B). To analyze the detected buggy cases for all firmware images quickly, we extended BASESPEC's L3 decoder identification (§IV-C) to find the erroneous functions discovered in §VII-C. By comparing their results, we observed the following interesting points:

When comparing the latest images with the oldest ones, we perceived that most of the identified cases have existed from the old days. For example, E8 and E9 are long-lived vulnerabilities from the oldest firmware in our dataset, and they are likely to have existed from earlier models. Moreover, some device models have the same mismatches and vulnerabilities. For example, *Model D*, *E*, *F*, and *G* have the same results, while those of *Model H* and *I* are the same. In addition, *Model C* and *B* show the same results. This result implies that the manufacturer may share the same/similar code base for those group of device models.

Furthermore, as the build dates of the oldest images are spread over four years, we noticed that there have been at least two security changes in the baseband implementation. This is because E6 newly appeared between April 2016 and April 2017, and E7 disappeared between March 2018 and February 2018. By analyzing them, we identified that E6 appeared with changes in GMM handlers, and E7 disappeared because of the addition of more security checks in EMM handlers. Meanwhile, there was no change in *Model H* and *I* in both mismatches and error cases except for E7 in their latest versions; however, the build date had a four-year gap. Thus, they were not affected by E6, which is a newly introduced error.

E. Applying BASESPEC to Other Vendors

To show that BASESPEC is applicable to other vendors, we analyzed three firmware images from $\texttt{Vendor}_2$, which is another one of the top 3 baseband chipset vendors. Currently, we only applied the syntactic analysis part (§V-C) of BASESPEC, which is sufficient to show its applicability to other vendors. We leave the semantic analysis as a future work because our underlying symbolic execution engine, angr, is still insufficient to model various library functions required to analyze the firmware from $\texttt{Vendor}_2$. Although we applied only a syntactic part of BASESPEC for $\texttt{Vendor}_2$, we discovered numerous mismatches and even a buffer overflow bug, which was previously unknown. Thus, we reported all the findings to the vendor.

TABLE VI: Summary results of syntactic comparison for $\texttt{Vendor}_2$

				Missing		Unknown		Invalid	
Model	Build Date	Msgs	IEs	i-IE	n-IE	i-IE	n-IE	i-IE	n-IE
Model X	Sep/2017	87	625	0	127	8(8)*	0	11(2)*	28
Model Y	Aug/2017	87	625	0	127	8(8)*	0	11(2)*	28
Model Z	Oct/2016	87	604	0	148	8(8)*	0	17(2)*	33

* Numbers in the parenthesis are the number of false positives.

Firmware acquisition. In the case of $\texttt{Vendor}_2$, there is no third party website that provides a well-structured list of firmware images, unlike the case of $\texttt{Vendor}_1$. Therefore, we collected firmware images using a web search. Among them, we selected three images based on the ARM architecture. $\texttt{Vendor}_2$ adopted the MIPS architecture instead of ARM in their recent devices since 2017. As BASESPEC currently supports ARM only, we used the old images. However, we found that recent MIPS devices are not largely different from the old ones, except for the architecture. In addition, we manually verified that the latest firmware still has the buffer overflow that we found in the old images. We believe that BASESPEC can apply to the recent firmware if it supports MIPS.

Firmware analysis. We first followed similar steps that we performed to the $\texttt{Vendor}_1$'s firmware (§IV) to uncover the obscurity of the $\texttt{Vendor}_2$'s firmware. Unlike $\texttt{Vendor}_1$, the firmware image includes a file that stores debug symbols; the file has a list of names and addresses of functions. By leveraging this file, we could recover function symbols in the firmware. Then, we identified a decoder function using the symbols and other debug messages in the firmware. Similar to the firmware of $\texttt{Vendor}_1$, that of $\texttt{Vendor}_2$ also implements a single decoder function to process various L3 messages. Finally, we figured out binary-embedded message structures from the decoder function.

We found that $\texttt{Vendor}_2$'s firmware also has a machine-friendly message structure, thereby allowing us to apply BASESPEC successfully. However, its format is completely different from $\texttt{Vendor}_1$'s. Instead of the hierarchical lists in $\texttt{Vendor}_1$ (Figure 4), the embedded structure in $\texttt{Vendor}_2$ maintains bytecode for each message, which comprises various simple opcodes. In particular, the firmware decodes a certain message by interpreting the corresponding bytecode. Several opcodes exist to handle different types of IEs. For instance, the `unpack_BITS` opcode parses bit-level IEs, and the `unpack_MAXBYTES` opcode parses IEs while limiting their maximum lengths. Moreover, the bytecode also has opcodes for control flow; for example, the `IF` opcode is used for checking IEI, and the `CALL` opcode can reuse other opcodes. Despite the different design of $\texttt{Vendor}_2$, it still meets our key intuitions (§III-B); we were able to apply BASESPEC to their firmware by implementing an interpreter for syntactic comparison and reusing other specification-related components.

Identifying mismatches. Table VI shows the number of mismatches that the syntactic comparison of BASESPEC found from three firmware images from $\texttt{Vendor}_2$. Notably, *Model X*

and *Model Y* showed the same result; although they are different models, they have the same embedded message structures. BASESPEC reported 8 unknown mismatches in all models; however, we identified that these are false positives. Specifically, the firmware arbitrarily divides a certain IE (e.g., `Progress Indicator` IE) into multiple pieces for its parsing. BASESPEC identified the separated pieces as different IEs and reported them as unknown mismatches. This issue also produced two false positives that resulted in invalid mismatches of each model. In summary, BASESPEC correctly reported 37 invalid mismatches from both *Model X* and *Y*, and 48 invalid mismatches from *Model Z*, with 10 false positives in each model.

Discovering bugs. By checking the invalid mismatches reported by BASESPEC, we found a buffer overflow vulnerability in all three models. Notably, although the firmware images appeared several years ago, the identified bug has been unknown previously. We found the bug by analyzing the mismatches from $\texttt{IE}_1$ and $\texttt{IE}_2$; these IEs belong to the $\texttt{MESSAGE}_1$ in the CS protocol.[4] According to the specification, the $\texttt{IE}_1$ can be 14 bytes long, and $\texttt{IE}_2$ can be 30 bytes long. However, BASESPEC reported that their lengths are swapped; that is, the decoder accepts the $\texttt{IE}_1$ with 30 bytes and the $\texttt{IE}_2$ with 14 bytes. This issue causes a buffer overflow in a handler function. When the handler copies incoming IEs to its internal structure, the $\texttt{IE}_1$ can overflow and overwrite a length field in the structure. Then, this corrupted structure will be delivered to other functions via internal messaging procedure. Therefore, in further handling, the overwritten length field can cause other security issues, such as denial of service. Additionally, other invalid mismatches revealed minor mistakes, *i.e.*, one or two-byte differences in the lengths, which can cause functional errors.

VIII. SECURITY ANALYSIS AND CASE STUDY

This section details interesting ones from the 9 erroneous cases (E1–E9). For the details of the other bugs, please see §A.

E1: Incorrect indices in Global IE List. By analyzing the mismatches from BASESPEC, we discovered that the baseband firmware is not compliant with specification because its `Global IE List` has incorrect indices for several IEs in ESM messages. As shown in Figure 4, the baseband manages IE information in the `Global IE List` to reuse it for multiple messages; various messages can share the same IEs. Each IE has a unique index that is assigned during implementation (i.e., `enum`). The firmware stores a certain IE's information in `Global IE List` according to the index, and then, the `Msg IE List` uses the indices to represent IEs contained in the message.

However, a few IEs used in the ESM protocol located incorrect indices in the `Global IE List` and they cause mismatches, as shown in Table VII. For example, ESM messages that contain `Header compression configuration` IE references index of 457 for the IE. However, the actual index of the IE is 456 in the global list; the `Control plane only indication` IE is located in 457, instead of `Header compression configuration` IE. It leads to missing and unknown mismatches in BASESPEC because the intended IE (i.e., `Header compression configuration`) will be missing in the message, and the unknown one (i.e., `Control plane only indication`) will appear. Developers confuse these indices due to the difficulties to correctly implement complex specifications of the baseband. We found this bug because BASESPEC can find mismatches from such inconsistency systematically.

TABLE VII: Inconsistent indices of IEs in the ESM protocol that appear in `Msg IE List` and in `Global IE List`.

IE name	Index in `Msg IE List`	Index in `Global IE List`
Header compression configuration	457	456
Control plane only indication	458	457
User data container	459	458
Release assistance indication	461	459
Extended protocol configuration options	456	460
Serving PLMN rate control	460	461

E3: Forgotten imperative IEs in the RR protocol. Further, we found that the `Feature Indicator` IE included in two RR messages, `IMMEDIATE ASSIGNMENT EXTENDED` and `IMMEDIATE ASSIGNMENT REJECT`, is not properly decoded in the current baseband implementation. This is buggy because the imperative IE is a mandatory field and should be handled. We found that this field was only a placeholder for alignment, which is called `Spare half octet`, in the old specification. However, from v10.4.0 released in Oct. 2011, the field was changed into a new imperative IE, named `Feature Indicator`. Nevertheless, the firmware fails to reflect this change and leaves this field without decoding. BASESPEC successfully detect this bug as missing mismatches for imperative IEs, as listed in Table V.

E6: Stack overflow in the GMM protocol. We found that the firmware has a stack buffer overflow in handling the GMM protocol, which can lead to remote code execution. In particular, the firmware accepts an `LV`-formatted IE called `Allocated P-TMSI` for the `P-TMSI reallocation command` message, which is a message in the GMM protocol. The length of this IE is defined as a fixed size of *5 bytes* in the specification. However, we found that the firmware in fact permits a variable-length input up to *255 bytes*, from the mismatches reported by BASESPEC. Then, we manually investigated its handler to understand the implication of this broken compliance.

Consequently, we found that the handler is vulnerable to a stack-based buffer overflow as shown in Figure 8. The developer blindly assumes the IE's length from the specification and copies its data to a 5-byte fixed buffer; however, the size can be larger than the one from the specification, up to 255 bytes. Notably, the `get_ie_bytes` function checks the lengths of certain IEs, as shown in lines 20–24 in Figure 8. However, the `Allocated P-TMSI` IE is not validated in the routine. Moreover, this buffer is located at 52 bytes apart from the function's return address, and the firmware has no stack protection techniques such as a stack canary. Therefore, attackers can hijack its control by overwriting the return address, and they can execute the arbitrary code using return-oriented programming [53]. We received an acknowledgement for the exploitability of this vulnerability from the vendor. Further, we found that all models except *Model H* and *Model I* are vulnerable. The two unaffected models retrieve the IE in different ways, using a hard-coded length of 5 bytes. Therefore, we can infer that this bug appeared from the updates between *Model G* and *Model H*.

E8: Integer underflow in the EMM protocol. From the BASESPEC's false positives, we found that the decoder handles the `SECURITY MODE COMMAND` message in the EMM protocol exceptionally. In particular, BASESPEC reported four unknown mismatches for the imperative IEs in the message. While

[4] We redacted the names because the bug is not patched yet by the vendor.

```
1  // We arbitrarily named functions and variables
2  // because they are stripped in the firmware
3  void handle_ptmsi_rellocation()
4  {
5      char allocated_ptmsi[5];
6      ...
7      get_IE_bytes(allocated_ptmsi,
8                   ALLOCATED_PTMSI_IDX);
9      ...
10 }
11
12 void get_ie_bytes(char *buf, enum IE_IDX idx)
13 {
14     int length;
15     char *value;
16
17     // Get a length of the IE in the message (Controllable)
18     length = get_ie_length(idx);
19
20     // Check lengths for certain IEs
21     if(idx == PLMN_LIST_IDX && length > 45)
22         length = 45;
23     if(idx == LSA_ID_IDX && length > 3)
24         length = 3;
25
26     // Get a value of the IE (Controllable)
27     value = get_ie_value(idx);
28     memcpy(buf, value, length);
29 }
```

Fig. 8: Code snippet of a stack buffer overflow vulnerability (E6)

investigating these results, we analyzed a caller of the decoder function, and Figure 9 shows its code snippet. This function specifies a body of a given message and calls the `l3_decoder` function, which is the BASESPEC's analysis target (Line 30). Notably, this function specially handles an incoming message, of which type is `SECURITY MODE COMMAND` (Lines 14–26); it invokes the decoder function with additional fields as depicted in Lines 5–11. We believe that the developers decided to reuse an existing decoding routine for IEs to parse these fields (e.g., MAC). Therefore, they embedded these additional fields as an IE form in the message structure, which result in unknown mismatches. Although these mismatches are false positives, they hint at the specialty of the `SECURITY MODE COMMAND` message, which is worthwhile to analyze manually. Note that such exceptional cases often involve misbehavior.

Interestingly, we noticed that the additional routine for the `SECURITY MODE COMMAND` is vulnerable to an integer underflow bug. This bug can cause a buffer overflow and even be exploited for remote code execution. The function copies a part of the message to a global buffer variable for later use, such as MAC validation. The function copies `"length - 5"` bytes, while limiting the maximum length to 38 bytes (Line 16). However, because the `length` variable is defined as a `short` type, `"length - 5"` can trigger underflow to a negative value. For example, the value can be `-1` if the `length` is `4`, and it is passed to the `memcpy` function (Line 22). Note that the `memcpy` function assumes the given length as an unsigned short type; hence, it would treat `-1` as `0xffff`. This makes it copy an abnormally large data, which produces a buffer overflow. We found that the overflowed data overwrites other variables including several function pointers. Therefore, using this vulnerability, an attacker can execute arbitrary code by overwriting the function pointers. We also received acknowledgement for the exploitability of this vulnerability from the vendor. We found that all the models are affected by this vulnerability regardless of the build date.

IX. DISCUSSION

Automating bug discovery. Although BASESPEC automatically identifies mismatches that are non-compliant with the specification, it requires additional efforts to discover bugs: one needs to analyze a few message handlers affected by the mismatches. This is because BASESPEC aims to pinpoint erroneous points by comparing baseband implementation based on cellular

```
1  // We arbitrarily named functions and variables
2  // because they are stripped in the firmware
3  void preprocess_emm(L3Msg *msg, short length)
4  {
5      // Figure out l3_body from msg
6      // e.g., msg->payload =
7      //  [Header(1 byte)][MAC(4)][SEQ(1)][L3 Header(2)][L3 Body(n)]
8      //    -> SECURITY_MODE_COMMAND:
9      //          l3_body = [MAC][SEQ][L3 Header][L3 Body]
10     //    -> Others:
11     //          l3_body = [L3 Body]
12     char *l3_body;
13     if (msg->type == SECURITY_MODE_COMMAND) {
14         // length - 5 can be underflown (integer underflow)
15         if (length - 5 > 38)
16             memcpy(global_buf, msg->payload + 5 , 38);
17         else {
18             // memcpy triggers buffer overflow
19             // (e.g., if length == 4, length - 5 = 0xffff)
20             memcpy(global_buf, msg->payload + 5,
21                         (unsigned short) length - 5);
22         }
23         l3_body = msg->payload + 1;
24     }
25     else
26         l3_body = msg->payload + 8;
27     l3_decoder(l3_body);
28     ...
29 }
```

Fig. 9: Code snippet of a integer underflow bugs (E8)

specifications (§V). Meanwhile, BASESPEC can collaborate with other promising techniques for full automation. From the analysis points identified by BASESPEC, fuzzing strategies [16], [52], [70] or hybrid analysis approaches [59], [68] can be applied. Further, recent emulation-based approaches [42], [27], [71] can co-operate with BASESPEC as well. We leave such promising improvements as a future work.

Applicability of our approach. BASESPEC addresses and leverages two natural characteristics of the baseband modem. First, as a real-time embedded system, a baseband initializes its memory layout in the booting procedure and has many interrupt routines that indirectly call functions; this severely hinders identifying the basic structures of the firmware. BASESPEC addressed this challenge with two preprocessing techniques (§IV-B). Second, as a network system, a baseband has a message decoder that utilizes embedded message structures to parse incoming messages. BASESPEC leverages this property and extracts binary-embedded message structures for a comparative analysis based on specifications (§IV-D). As other baseband modems would share these properties, we believe that BASESPEC's approach can be applicable although it requires considerable manual efforts for analyzing the firmware once.

In addition, BASESPEC currently supports standard L3 messages, but other cellular protocols can be analyzed similarly. This is because by design, the protocol specifications should describe every message structure for cellular protocols in consistent forms [3]. These systematic forms allow BASESPEC to automatically analyze these messages. Therefore, we believe BASESPEC can be applied to other protocols that have well-structured messages defined in ASN.1 or CSN.1, such as the messages for the radio resource control protocol.

Towards other types of bugs. Firmware analysis for baseband inherits the fundamental challenges of binary analysis. For example, discovering service-related bugs such as bypassing security channels [39] is extremely difficult. This is because a baseband implements numerous cellular protocols that have convoluted states; therefore, various stateful information should be considered in the analysis. Moreover, building a reference

for logical bugs from the specifications is also not trivial [34], [35], [10], [33], [8]. Therefore, we invite future research in this field by introducing BASESPEC as an entry point. Although BASESPEC currently cannot cover logical bugs, its ultimate goal is to build such a model and discover logical bugs with a comparative analysis.

X. RELATED WORK

A. Studies on cellular baseband software

A baseband processor in cellular devices plays an important role in cellular communication. Thus, researchers have studied the security of protocol implementations in a BP [47], [46], [63], [64], [25], [15], [42]. Early stage of these approaches focused on GSM networks, especially SMS or cell broadcast messages [47], [46], [63]. Without directly analyzing the firmware of baseband, these approaches implemented a fuzzer based on the SMS message structure, and they found several memory-related bugs that force the device to crash or behave abnormally.

Other studies [64], [25], [15] focused on analyzing baseband firmware for layer 3 protocols. Weinmann [64] showed a practical approach to analyze memory-related bugs in GSM protocol stacks in baseband. Golde *et al*. [25] and Cama [15] analyzed recent Exynos firmware utilizing memory dumps. Notably, they discovered RCE 0-days and were rewarded at Mobile Pwn2Own. Although these approaches yield promising insights into baseband analysis, they have a limitation in that they require the repetitive manual analysis. To address this limitation, Maier *et al*. [42] recently proposed an emulation-based analysis on the RRC/EMM protocols. To emulate functions related to those protocols, they manually analyzed MediaTek firmware and hooked all related functions. Then, they ran a famous fuzzer, AFL++ [43], and found a heap overflow.

However, these studies also suffer from two fundamental limitations as described in §III-A: 1) it is difficult to create an oracle for identifying a bug, and 2) they are not applicable to diverse models/versions. BASESPEC overcomes these limitations by leveraging a comparative analysis based on specifications (§III-B). With a prototype of BASESPEC, we discovered many erroneous cases including 2 0-days for RCE (§VII). Moreover, BASESPEC is applicable to diverse models/versions (§VII-D).

Meanwhile, other studies focused on the AT interface of the baseband [29], [44], [60], [37], [45]. Notably, Tian *et al*. [60] conducted a comprehensive analysis on over 2,000 Android devices across 11 vendors using firmware images and found multiple bugs. Lastly, various studies provide useful insights for reverse-engineering the baseband [65], [21], [28], [11], [69].

B. Studies on cellular network protocols

Numerous approaches rely on dynamic analysis for inspecting over-the-air bugs on cellular networks, utilizing open-source cellular stacks [26], [14], [48], [66] and low-cost software-defined radios (SDRs) [22], [49]. These approaches do not need to directly analyze baseband firmware; instead they need a custom test-bed. By sending crafted messages to target devices, they discovered several service-related bugs in the L3 protocols that affect service quality [51], [41] or leak private information [56], [57]. Recent approaches [61], [54], [23], [39] have attempted to minimize the manual efforts involved in the dynamic analysis, leveraging abnormal messages. Some other studies focused on different layers, protocols, or domains on cellular networks such as VoLTE [38], SS7/Diameter [32], uplink messages [39], [18], or lower layers [55], [40], [67].

These approaches are advantageous for finding service-related bugs on specifications instead of software bugs such as memory corruption. Nevertheless, they require real hardware and considerable domain knowledge for cellular networks and specifications. Therefore, testing every implemented messages on diverse devices or building a bug oracle requires huge efforts. BASESPEC tackles this by directly matching the documented specification with the binary-embedded one, thereby discovering hundreds of mismatches that may break the compliance with the specification on multiple devices (§VII-C). Currently, BASESPEC cannot discover bugs identified by these studies; however, its methodology can be extended as discussed in §IX.

XI. CONCLUSION

In this study, we conducted the first systematic comparative analysis of cellular baseband software and specifications. By leveraging the natural characteristics of the baseband as a real-time embedded device and a network modem, we designed BASESPEC that automatically extracts binary-embedded specification and compares it with the documented one syntactically and semantically. By running an automated prototype of BASESPEC on 18 baseband firmware images from one of the top three vendors, we discovered hundreds of mismatches that are non-compliant with the specification. By analyzing the mismatches, we discovered a total of 9 bugs, of which 5 are functional errors and 4 are memory-related vulnerabilities including two critical RCE 0-days. Further, we applied BASESPEC to another vendor and discovered several mismatches, two of which cause a buffer overflow bug.

ACKNOWLEDGMENT

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00831, a study on physical layer security for heterogeneous wireless network)

REFERENCES

[1] 3GPP, "3GPP Partners," https://www.3gpp.org/about-3gpp/partners.

[2] ——, "3GPP Releases," https://www.3gpp.org/specifications/releases.

[3] ——, "TS 24.007; Mobile radio interface signalling layer 3; General aspects," 2018.

[4] ——, "TS 24.008; Mobile radio interface Layer 3 specification; Core network protocols; Stage 3," 2019.

[5] ——, "TS 24.011; Point-to-Point (PP) Short Message Service (SMS) support on mobile radio interface," 2019.

[6] ——, "TS 24.301; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3," 2019.

[7] ——, "TS 44.018; Mobile radio interface layer 3 specification; GSM/EDGE Radio Resource Control (RRC) protocol," 2019.

[8] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New Privacy Issues in Mobile Telephony: Fix and Verification," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[9] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[10] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A Formal Analysis of 5G Authentication," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[11] D. Berard and V. Fargues, "How to design a baseband debugger," in *Information and Communication Technology Security Symposium (SSTIC)*, Rennes, France, jun 2020.

[12] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Budapest, Hungary, Mar.–Apr. 2008.

[13] S. Bugrara and D. Engler, "Redundant state detection for dynamic symbolic execution," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, San Jose, CA, Jun. 2013.

[14] D. A. Burgess and H. S. Samra, "The OpenBTS Project," *Open Source Cellular Infrastructure*, 2008. [Online]. Available: http://openBTS.org

[15] A. Cama, "A walk with Shannon," in *OPCDE*, 2018.

[16] S. K. Cha, M. Woo, and D. Brumley, "Program-Adaptive Mutational Fuzzing," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015, pp. 725–741.

[17] S. Chin, "Top-tier smartphone makers going to in-house processors: report," https://www.fierceelectronics.com/electronics/top-tier-smartphone-makers-going-to-house-processors-report.

[18] M. Chlosta, D. Rupprecht, T. Holz, and C. Pöpper, "LTE Security Disabled: Misconfiguration in Commercial Networks," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Miami, FL, May 2019.

[19] E. Coppa, D. C. D'Elia, and C. Demetrescu, "Rethinking Pointer Reasoning in Symbolic Execution," in *Proceedings of the 32rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct. 2017.

[20] A. Costin, A. Zarras, and A. Francillon, "Towards Automated Classification of Firmware Images and Identification of Embedded Devices," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2017, pp. 233–247.

[21] G. Delugré, "Reverse engineering a Qualcomm baseband," in *28th Chaos Communication Congress*, Berlin, Germany, dec 2011.

[22] U. Ettus, "B210." [Online]. Available: https://www.ettus.com/all-products/ub210-kit/

[23] K. Fang and G. Yan, "Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning," in *Proceedings of the 23th European Symposium on Research in Computer Security (ESORICS)*, Barcelona, Spain, Sep. 2018.

[24] Glyph & Cog, LLC, "pdftotext(1) - Linux man page," https://linux.die.net/man/1/pdftotext.

[25] N. Golde and D. Komaromy, "Breaking Band: reverse engineering and exploiting the shannon baseband," *REcon*, 2016. [Online]. Available: https://comsecuris.com/slides/recon2016-breaking_band.pdf

[26] I. Gomez-Miguelez, A. Garcia-Saavedra, P. D. Sutton, P. Serrano, C. Cano, and D. J. Leith, "srsLTE: An Open-Source Platform for LTE Evolution and Experimentation," in *Proceedings of the 10th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization (WiNTECH)*, New York City, NY, Oct. 2016.

[27] N. Group *et al.*, "A linux system call fuzzer using TriforceAFL," https://github.com/nccgroup/TriforceAFL, 2017.

[28] W. Hengeveld, "IDA processor module for the hexagon (QDSP6v55) processor," https://github.com/gsmk/hexagon, 2013.

[29] C. Heres, A. Etemadieh, M. Baker, and H. Nielsen, "Hack all the things: 20 devices in 45 minutes," in *DEFCON*, 2014.

[30] S. Hex-Rays, "IDA FLIRT Technology: In-Depth," https://www.hex-rays.com/products/ida/tech/flirt/in_depth/.

[31] ——, "IDA: Hex-Rays," https://www.hex-rays.com/products/ida.

[32] S. Holtmanns, S. P. Rao, and I. Oliver, "User Location Tracking Attacks for LTE Networks Using the Interworking Functionality," in *Proceedings of the 15th International Federation for Information Processing (IFIP) Networking Conference*, Vienna, Austria, May. 2016.

[33] X. Hu, C. Liu, S. Liu, W. You, Y. Li, and Y. Zhao, "A Systematic Analysis Method for 5G Non-Access Stratum Signalling Security," *IEEE Access*, vol. 7, pp. 125 424–125 441, 2019.

[34] S. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[35] S. R. Hussain, M. Echeverria, I. Karim, O. Chowdhury, and E. Bertino, "5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[36] M. Jiang, Y. Zhou, X. Luo, R. Wang, Y. Liu, and K. Ren, "An empirical study on arm disassembly tools," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Online, Jul. 2020.

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

[38] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim, "Breaking and Fixing VoLTE: Exploiting Hidden Data Channels and Mis-implementations," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[39] H. Kim, J. Lee, E. Lee, and Y. Kim, "Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[40] M. Lichtman, R. P. Jover, M. Labib, R. Rao, V. Marojevic, and J. H. Reed, "LTE/LTE-A Jamming, Spoofing, and Sniffing: Threat Assessment and Mitigation," *IEEE Communications Magazine*, vol. 54, no. 4, pp. 54–61, 2016.

[41] H. Lin, "LTE REDIRECTION: Forcing Targeted LTE Cellphone into Unsafe Network," in *Hack In The Box Security Conference (HITBSecConf)*, 2016.

[42] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband SAnitized Fuzzing through Emulation," in *Proceedings of the 13th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Virtual, Jul. 2020.

[43] Marc Heuse, Heiko Eißfeld, Andrea Fioraldi, and Dominik Maier, "AFLplusplus (AFL++)," https://github.com/vanhauser-thc/AFLplusplus, 2020.

[44] L. Miras, "Baseband playground," in *Proceedings of the 7th Ekoparty Security Conference*, Buenos Aires, Argentina, Sep. 2011.

[45] G. Miru, "Path of Least Resistance: Cellular Baseband to Application Processor Escalation on Mediatek Devices," https://comsecuris.com/blog/posts/path_of_least_resistance/, 2017.

[46] C. Mulliner, N. Golde, and J.-P. Seifert, "SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale." in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[47] C. Mulliner and C. Miller, "Fuzzing the Phone in your Phone," in *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, Jul. 2009.

[48] N. Nikaein, R. Knopp, F. Kaltenberger, L. Gauthier, C. Bonnet, D. Nussbaum, and R. Ghaddab, "OpenAirInterface: An Open LTE Network in a PC," in *Proceedings of the 20th Annual international conference on Mobile computing and networking (MobiCom)*, Maui, Hawaii, Sep. 2014.

[49] L. Nuand, "bladeRF." [Online]. Available: https://www.nuand.com/bladerf-2-0-micro/

[50] D. A. Ramos and D. Engler, "Under-Constrained Symbolic Execution: Correctness Checking for Real Code," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[51] M. T. Raza, F. M. Anwar, and S. Lu, "Exposing LTE Security Weaknesses at Protocol Inter-Layer, and Inter-Radio Interactions," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 312–338.

[52] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing Seed Selection for Fuzzing," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[53] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, Mar. 2012.

[54] D. Rupprecht, K. Jansen, and C. Pöpper, "Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[55] D. Rupprecht, K. Kohls, T. Holz, and C. Pöpper, "Breaking LTE on Layer Two," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[56] A. Shaik, R. Borgaonkar, N. Asokan, V. Niemi, and J.-P. Seifert, "Practical Attacks Against Privacy and Availability in 4G/LTE Mobile Communication Systems," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[57] A. Shaik, R. Borgaonkar, S. Park, and J.-P. Seifert, "New vulnerabilities in 4G and 5G cellular access network protocols: exposing device capabilities," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Miami, FL, May 2019.

[58] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[59] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[60] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace, and K. R. Butler, "Attention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[61] G.-H. Tu, Y. Li, C. Peng, C.-Y. Li, H. Wang, and S. Lu, "Control-Plane Protocol Interactions in Cellular Networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 223–234, 2014.

[62] "URL Anonymized due to vendor's request."

[63] F. Van Den Broek, B. Hond, and A. C. Torres, "Security Testing of GSM Implementations," in *International Symposium on Engineering Secure Software and Systems (ESSoS)*. Springer, 2014, pp. 179–195.

[64] R.-P. Weinmann, "Baseband Attacks: Remote Exploitation of Memory Corruptions in Cellular Protocol Stacks," in *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, Bellevue, WA, Aug. 2012.

[65] ——, "Baseband exploitation in 2013: Hexagon challenges," in *PACSEC 2013*, Tokyo, Japan, 2013.

[66] B. Wojtowicz, "OpenLTE," *An open source 3GPP LTE implementation*, 2016. [Online]. Available: http://openlte.sourceforge.net

[67] H. Yang, S. Bae, M. Son, H. Kim, S. M. Kim, and Y. Kim, "Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[68] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018, pp. 745–761.

[69] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[70] M. Zalewski, "American fuzzy lop (AFL)," http://lcamtuf.coredump.cx/afl, 2017.

[71] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

APPENDIX A
CASE STUDIES ON OTHER MIS-IMPLEMENTATIONS

E2: Redundant IEs in the SS protocol. We found that several messages in the SS protocol have redundant IEs, thereby resulting in unknown mismatches. We believe that it happens because of blind copy-and-paste; a developer seems to copy the structure of the `RELEASE COMPLETE` message in the *CS* protocol to implement the `REGISTER` and `RELEASE COMPLETE` messages in the *SS* protocol. They are almost same; however, the `RELEASE COMPLETE` message in the CS protocol has an additional IE called `User-user`, which is unspecified in the other two messages. Such a redundant IE makes the firmware non-compliant with the specification. In addition, it can cause unintentional behaviors in the firmware because its handler can receive unexpected messages. We found this mistake in all models in our dataset.

E4: Incorrect IEI value in the EMM protocol. We also discovered that the `NAS message container` IE of the `Control plane service request` message in the EMM protocol has an incorrect IEI, thereby resulting in both missing and unknown mismatches. Because there is only one missing mismatch and one unknown mismatch with the same length, we can determine that the incorrect IEI value is the root cause of those mismatches. The IE should have `0x67` as its IEI, but `0xff` is stored in the firmware. In fact, the `NAS message container` IE is used in other messages as imperative IEs, which do not require an IEI value. However, when the `Control plane service request` message is added in the specification of version v13.6.1, which is released in August 2016, the IE is first used as a non-imperative IE with the IEI of `0x67` in the message. Therefore, this mismatch implies that developers missed changing the IEI when adding this new message. Since *Model H* and *Model I* do not have the message, they are not affected by this mistake.

E5: Unknown IE in the GMM protocol. We discovered that the `Routing area update accept` and `Attach accept` messages in the GMM protocol share one unknown non-imperative IE, whose IEI is `0xB` and length is `2` bytes. These messages have many unimplemented yet non-imperative IEs. We believe that this buggy IE is one of them with a misconfigured IEI. This error affects all models except *Model H* and *Model I*.

E7: Buffer overflow in the EMM protocol. Similar to E6, the handler function for the `EMM information` message in the EMM protocol mishandled the `Network daylight saving time` IE. In the 6 oldest firmware images in our dataset, the handler copies this IE into a global buffer without checking its length. This buffer overflow can corrupt nearby configuration variables and cause unintended behaviors. We found that recent firmware contains a routine to limit its length to 7 bytes at maximum.

E9: Invalid pointer dereference in the CS protocol. We found invalid pointer dereference while analyzing `START DTMF ACKNOWLEDGE` message in the CS protocol. To decode this message, the L3 decoder function first initializes a pointer with `-1` and later sets it to other data. If the message has invalid IEs, this pointer can hold the initial value (i.e., `-1`) until dereferencing. Therefore, the decoder should check whether the pointer has a valid address before dereferencing. Unfortunately, it fails to check the pointer properly, and it compares the pointer with `NULL` instead of `-1`. Therefore, the firmware accesses memory at `-1`, which results in access violation. This bug is discovered in all models (i.e., from Model A to Model I).