BASECOMP: A Comparative Analysis for Integrity Protection in Cellular Baseband Software

Eunsoo Kim* *KAIST*
Min Woo Baek* *KAIST*
CheolJun Park *KAIST*
Dongkwan Kim *Samsung SDS*
Yongdae Kim *KAIST*
Insu Yun *KAIST*

Abstract

Baseband software is an important component in cellular communication. Unfortunately, it is almost impossible to implement baseband software correctly due to the complexity and the large volume of cellular specifications. As a result, dynamic testing has been widely used to discover implementation bugs in them. However, this approach suffers from the reachability problem, resulting in many missed bugs. Recently, `BaseSpec` proposed a static approach for analyzing baseband. However, `BaseSpec` requires heavy manual analysis and is limited to message decoding, failing to support integrity protection, the most critical step in mobile communication.

In this paper, we propose a novel, semi-automated approach, BASECOMP, for analyzing integrity protection. To tame the complexity of baseband firmware, BASECOMP utilizes probabilistic inference to identify the integrity protection function. In particular, BASECOMP builds a factor graph from the firmware based on specifications and discovers the most probable function for integrity protection. Then, with additional manual analysis, BASECOMP performs symbolic analysis to validate that its behavior conforms to the specification and reports any discrepancies. We applied BASECOMP to 16 firmware images from two vendors (Samsung and MediaTek) in addition to srsRAN, an open-source 4G and 5G software radio suite. As a result, we discovered 29 bugs, including a NAS AKA bypass vulnerability in Samsung which was assigned critical severity. Moreover, BASECOMP can narrow down the number of functions to be manually analyzed to 1.56 on average. This can significantly reduce manual efforts for analysis, the primary limitation of the previous static analysis approach for baseband.

1 Introduction

Baseband software is critical in smartphones, one of the most important devices these days, as it enables cellular communication. Despite its importance, unfortunately, it is extremely challenging to implement the baseband software correctly. In particular, the baseband software should be compliant with the specification defined by the 3rd Generation Partnership Project (3GPP) [1]. This specification is written in natural languages by hand and contains hundreds of pages; therefore, it is nearly impossible to understand it comprehensively. More seriously, due to its complexity and volume, inconsistencies and ambiguities are frequently discovered in these documents. As a consequence, developers are prone to make errors when implementing baseband software, leading to serious security issues such as denial of service or even authentication bypass [19, 24, 29, 31, 35, 41, 44, 45, 48, 52, 54, 59].

To remedy this issue, researchers have proposed several approaches to discover implementation bugs in baseband software. To tame the complexity of baseband firmware, dynamic analysis is mainly utilized; it sends messages and observes responses from real devices [29, 31, 35, 44, 45, 48, 52, 54, 59] or emulated ones [19, 24, 41] to discover bugs. This dynamic method is effective in avoiding efforts to understand firmware details; however, due to the large search space consisting of cellular messages, it is inevitable to restrict the search space (e.g., assuming syntactic correctness), leading to missing bugs. Recently, `BaseSpec` [33] employs a static approach that combines manual analysis and comparative analysis to discover implementation bugs. Despite its success, `BaseSpec` is limited to analyzing only an early stage of baseband software — message decoding. As a result, it fails to analyze integrity protection, which is a core procedure in mobile communication, being the main target of many previous works [29, 30, 35, 48]. Moreover, it solely relies on human experts to gather data for its analysis, which requires significant manual effort.

In this paper, we propose BASECOMP, a new static analysis approach for analyzing integrity protection in baseband software. To support the large and complex baseband software, BASECOMP combines probabilistic inference and comparative analysis. First, BASECOMP identifies the integrity protection function using probabilistic inference. In more detail, BASECOMP builds a factor graph using specification-driven features and rank functions in baseband firmware to

*These two authors equally contributed.

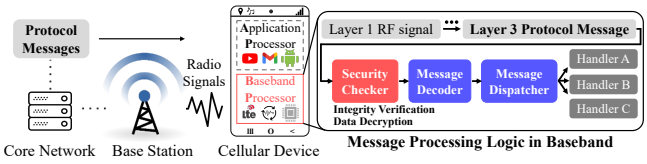


Figure 1: Overall cellular network architecture

discover the most probable function for integrity protection. Our evaluation shows that this approach is extremely effective; we discovered the integrity protection function within 1.56 candidates on average. It is worth noting that our technique is based on specification rather than implementation, making it available across multiple models and vendors. Once the integrity protection function has been identified, we rely on manual analysis to obtain several data for comparative analysis; we build a firmware-specific configuration and a vendor-specific module to support diverse firmware binaries. Finally, with the given data, BASECOMP performs a comparative analysis following the specification. In particular, BASECOMP symbolically analyzes the integrity protection function in the firmware to exhaustively examine messages that can be accepted as plaintext. Then, we compare them with the specification (TS 24.301 [3]) and discover mismatches.

We applied BASECOMP to 16 firmware binaries from Samsung, MediaTek devices, and srsRAN. As a result, we identified 34 mismatches in integrity protection, resulting in 29 bugs (8, 7, 14 in Samsung, MediaTek, and srsRAN respectively). In particular, we discovered several vulnerabilities that can lead to denial of service and information leakage. More importantly, we discovered a NAS AKA (authentication and key agreement) bypass vulnerability that affects a majority of devices that use Samsung baseband. To encourage further research, we open-source our prototype of BASECOMP at `https://github.com/kaist-hacking/BaseComp`.

In summary, this paper makes the following contributions:

- We propose BASECOMP, a novel approach for analyzing integrity protection in baseband software using probabilistic inference and comparative analysis.
- We applied BASECOMP to 16 firmware images from two vendors, Samsung, MediaTek, and srsRAN. As a result, we discovered 29 bugs, including one NAS AKA bypass vulnerability. We responsibly disclosed all bugs.
- To foster future research, we open-source our prototype of BASECOMP at `https://github.com/kaist-hacking/BaseComp`.

2 Background

Figure 1 illustrates an overview of the cellular network architecture and the message processing logic of a cellular baseband processor. The terminologies can vary according to the generation of cellular technology, but for the sake of simplicity, hereafter we use generic terms. Due to the nature of mobile specification, many acronyms are used, and those used in this paper are summarized at Appendix A.

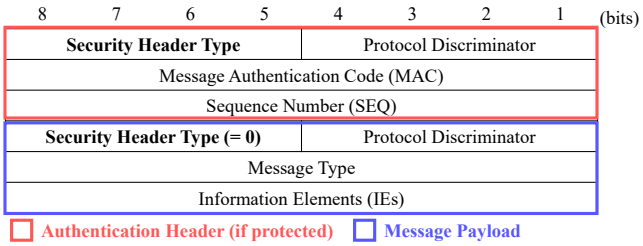


Figure 2: NAS message structure.

2.1 Cellular Network Architecture

Cellular networks primarily consist of three components: cellular devices, base stations, and a core network. Cellular devices and a core network exchange various cellular protocol messages through a base station. Each component has a dedicated behavior depending on its current status and received message, following the cellular specification.

2.2 Baseband Processor and Software

A cellular device, mostly a smartphone, includes two distinct processors: an application processor and a baseband processor. The application processor (AP) runs a mobile operating system such as Android or iOS, and the baseband processor (BP) manages cellular communication. To satisfy real-time requirements for processing radio signals and various protocol messages, the BP runs a real-time operating system as its firmware.

Baseband firmware is typically proprietary, and its implementation details are not publicly available. For example, among the top three mobile processors — Qualcomm's Snapdragon, MediaTek's Helio, and Samsung's Exynos [36], none of these manufacturers publish implementation details of their products, such as the source code or firmware structures. As a result, it is common practice to *manually* analyze baseband firmware to uncover software bugs or vulnerabilities [7, 16, 23, 24, 33, 61, 66].

2.3 Protocol Messages and Processing Logic

The cellular protocol stack follows the OSI paradigm as other wired networks do. The radio interface covers layers 1 and 2, and cellular core procedures are delivered at layer 3. The core procedures at layer 3 consist of various protocol messages for mobility/session management, call control, or user authentication. Among these protocols, Non-Access Stratum (NAS) is a collection of essential protocols in the communication between cellular devices and a core network, particularly for mobility and session management.

While implementation details vary by manufacturer, cellular devices share a common logic for processing protocol

Table 1: Possible security header types for downlink messages.

Value	Description
0	*Plain* NAS message, not security protected
1	Integrity protected
2	Integrity protected and ciphered
3	Integrity protected with new EPS security context
Others	For special purposes and reserved values

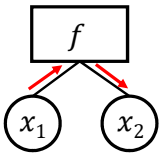
messages, which is illustrated on the right side of Figure 1. After processing the radio signals, the BP of a cellular device first checks whether the received message is *security protected*. This step, which is called *integrity protection*, includes verifying the message's integrity and optionally decrypting the message. Note that most cellular protocol messages should be security protected to determine whether they are actually sent from a legitimate core network. Next, the BP decodes the message using a pre-defined message structure [2]. It then performs an appropriate action with respect to each message, as stated in the specification [3].

2.4 Security Features and Message Structures

To guarantee secure communication between cellular devices and a core network, cellular protocols employ multiple security protection mechanisms such as encryption or integrity protection. When a cellular device joins a core network, it selects a protection mechanism for further secure communication and derives the keys required for protection using a master key shared priorly. However, it is unavoidable to use plaintext messages before a secure communication channel is established. To reduce attack vectors, the cellular specification permits only a few messages to be accepted in plaintext, while forcing all other messages without security protection to be rejected or discarded [3].

To accommodate this security feature, the structure of a NAS message is largely divided into two parts, namely an authentication header followed by a message payload as depicted in Figure 2. It is worth noting that messages without security protection do not have an authentication header. The authentication header consists of four fields: security header type, protocol discriminator (PD), message authentication code (MAC), and sequence number. In particular, the security header type represents the protection mechanism that is applied to the message as listed in Table 1. Therefore, the security header type of an incoming message is crucial for baseband firmware in determining whether to perform message integrity verification and decryption.

Following that, the message payload is composed of a security header type, a PD, a message type, and information elements (IEs). The message payload is essentially a plaintext NAS message without security protection; thus, its security header type has a fixed value of `0`. Meanwhile, this message can be encrypted and embedded as payload after the authentication header.



		$x_1 \xrightarrow{Pos} x_2$	$x_1 \xrightarrow{Neg} x_2$	$x_1 \xrightarrow{Pref} x_2$
x_1	x_2	f_{Pos}	f_{Neg}	f_{Pref}
0	0	0.5	0.5	0.5
0	1	0.5	0.5	p_{Pref}
1	0	p_{Pos}	$1-p_{Neg}$	$1-p_{Pref}$
1	1	$1-p_{Pos}$	p_{Neg}	0.5

Figure 3: A factor graph with three different functions: a positive relationship, a negative relationship, and a preferable relationship. Note that we assume that all parametric probability — namely, p_{Pos}, p_{Neg}, and p_{Pref} — are less than 0.5.

2.5 Probabilistic Inference

A factor graph is a type of probabilistic graph model. There are two different types of nodes in the factor graph: a variable node and a function node. A variable node represents a random variable, whereas a function node contains a relationship between variable nodes that are connected to it. In a factor graph, edges can only connect a variable node and a function node, i.e., a factor graph is a bipartite graph [40].

The factor graph can be used for probabilistic inference. If we have multiple pieces of probabilistic information, we can build a factor graph to determine the probability of our consequence. By defining functions, we can represent various relationships between variable nodes. Figure 3 shows examples of such various relationships. With a factor graph, we can represent a positive relationship between nodes (if x_1 is true, x_2 will be likely true), a negative relationship (if x_1 is true, x_2 will be likely false), and also a preference relationship (x_2 is more likely to be true than x_1). Then, we can calculate the marginal probability of a consequence using belief propagation for probabilistic inference [64].

3 Motivations

In this section, we discuss the limitations of existing work that motivate us to develop BASECOMP.

3.1 Limitations of Dynamic Testing

Motivation. Due to the extremely large volume (dozens of MBs) and complexity of baseband firmware, dynamic testing is frequently used in analysis [19, 24, 29, 31, 35, 44, 45, 48, 52, 54, 59]. This approach is effective as it avoids substantial efforts in understanding firmware; nevertheless, it often relies on domain-specific knowledge that may not be applicable in a specific implementation. In particular, to tame a large search space that comprises diverse messages, dynamic testing considers only a subset of them or assumes their syntactic correctness. However, this leads to missing bugs caused by unconsidered messages or syntactically broken ones (§7.4).

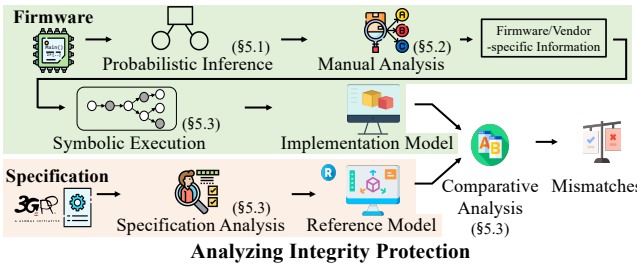


Figure 4: Workflow of BASECOMP.

Our approach: Static analysis. To address this issue, BASECOMP utilizes static analysis, which is always considered a complementary method for dynamic analysis. However, as demonstrated by the sparse use of static analysis approaches, it is non-trivial to apply static analysis to baseband firmware due to its large volume and obscurity. Still, static analysis is valuable; BASECOMP could detect 6 types of bugs that dynamic testing couldn't discover (§7.4). In particular, a NAS AKA bypass vulnerability in Samsung, which BASECOMP discovers, remained hidden even with many dynamic testing trials, as this issue could only be triggered with a syntactically invalid message. This message — a message with non-zero security header type without authentication header — violates the syntactic structure defined in the specification (clause 9.1 in [3]). This also leads to Wireshark, a packet analysis tool, failing to parse such a message. Moreover, in srsRAN, BASECOMP discovers a NAS AKA bypass vulnerability by setting the security header type to 4, which represents integrity protected and ciphered with new EPS security context, together with a dumb header. According to the specification, this security header type is only valid for uplink, making it out-of-scope in many blackbox testing tools [29, 35].

3.2 Limitations of `BaseSpec`: No Analysis for Logic Bugs

Motivation. Recently, `BaseSpec` [33] suggested a static analysis method for message decoding and successfully discovered multiple functional errors and critical security issues. However, `BaseSpec` fails to analyze integrity protection, which was the main target of many previous works [29, 30, 35, 48]. Even though memory corruption is extremely powerful and enables an adversary to obtain full privilege on a baseband processor, the exploitability is often unreliable and can be mitigated by a generic defense mechanism. For instance, we found that a recent Samsung baseband incorporates a stack canary[1] to mitigate stack overflows, which is one of the key issues uncovered by `BaseSpec`. On the contrary, a logical bug is straightforward to exploit and cannot be prevented by a generic solution. Due to its significance, many researchers have studied logical vulnerabilities [19, 29, 31, 35, 48, 52]; however, they are all constrained to a dynamic manner.

Our approach: Specification-driven comparative analysis. To uncover integrity protection issues in baseband, BASECOMP performs comparative analysis based on specification. Particularly, BASECOMP extracts symbolic constraints for plaintext messages that are permitted by a baseband's integrity protection function. Then, BASECOMP compares the firmware's constraints with those extracted from the specification in order to discover any inconsistencies. In contrast to dynamic testing, BASECOMP makes no assumptions such as message types or syntactic correctness throughout this analysis. As a result, BASECOMP can discover mismatches in the integrity protection function without having to reduce the search space.

3.3 Limitations of Manual Analysis

Motivation. As stated in §2.2, the implementation details of baseband firmware are not publicly available. Hence, considering the substantial number of functions contained in baseband firmware, inquiring for information to scale down the scope of analysis is highly resource-consuming.

Our approach: Probabilistic inference. Inspired by recent binary analysis work [67, 68], we devise a technique that locates the integrity protection function in firmware using probabilistic inference. Specifically, BASECOMP builds a factor graph according to the mobile specification, TS 24.301 [3], and discovers the most probable function that implements integrity protection. With this help, only an average of 1.56 out of 80K functions is needed to be analyzed for comparative analysis, significantly reducing manual efforts.

4 Overview

4.1 Workflow

Figure 4 illustrates BASECOMP's workflow. BASECOMP systematically analyzes the key component of baseband software, integrity protection. To investigate integrity protection in baseband software (§5), we begin with probabilistic inference that locates an integrity protection function in baseband software (§5.1). Then, we rely on manual analysis to build a firmware-specific and a vendor-specific model (§5.2). After that, BASECOMP symbolically analyzes the software to obtain symbolic constraints for plaintext messages that are allowed in baseband; this represents an implementation model (§5.3). Also, BASECOMP constructs a reference model for integrity protection by analyzing the specification. After that, BASECOMP compares the two models and reports any mismatches (§5.3). To identify bugs, we further analyze these mismatches and conclude their implications.

[1] Samsung applied stack canary from Galaxy S21 5G (G991) since its first release in January 2021.

4.2 Scope of This Work

Among the various protocols in the cellular network, we choose the EPS Mobility Management (EMM) protocol and its integrity protection as our target. This protocol contains a variety of messages and is critical to the cellular core network. As the EMM protocol has numerous complicated logic, including user identification, authentication, message encryption, and integrity checks, many previous studies analyzed the protocol as their target [28,29,35,48]. In addition, we support two of the top three baseband processor vendors, Samsung and MediaTek [36]. It is worth noting that we cannot support Qualcomm, the top vendor of baseband firmware, due to its proprietary architecture, Hexagon; unfortunately, most of our underlying tools (i.e., IDA pro and angr) do not support this architecture. To add, we also support srsRAN, a widely used open-source project that implements 4G and 5G software radio suites [21].

4.3 Threat Model

We assume an active attacker model on the wireless channel, the same model that has been widely used in previous cellular security researches [28,48,52,54]. In this model, an attacker can drop, intercept, modify, or inject messages between the base station and the victim. Also, we assume that the cryptographic keys are secure, that is an attacker can only produce plaintext messages or messages with the wrong MAC. To perform malicious conduct with these capabilities, an attacker can operate a fake base station with strong signals [15] or use SigOver attack [63]. In this work, the attacker will try to bypass the integrity protection of the EMM protocol and trigger logical bugs such as denial of service (§3.2).

5 Design

In this section, we discuss BASECOMP's approach to analyze integrity protection.

5.1 Probabilistic Inference for Integrity Protection Function

To analyze the baseband firmware's integrity protection function, we need to identify it first. For that, BASECOMP utilizes probabilistic inference based on specification. According to the subclause 4.4.4.2 of TS 24.301 [3], integrity protection should be implemented as follows:

> **[TS 24.301, Sec.4.4.4.2]** Except the messages listed below, no NAS messages shall be processed, unless the network has established secure exchange of NAS messages:
> - `Identity Request` if Identity Type is IMSI
> - `Authentication Request`, `Authentication Reject`, and `Detach Accept`
> - `Attach Reject`, `Tracking Area Update Reject`, `Service Reject` if the EMM cause is not #25
>
> Once the secure exchange has been established, the UE shall not process any NAS signaling messages unless they have been successfully integrity checked.

Following this specification, BASECOMP takes three steps in discovering the integrity protection function. Figure 5 shows an example of these steps — how BASECOMP constructs a factor graph for a given call graph. Note that each step is denoted by a grey box with a related label.

Step 1: Identifying MAC functions. To discover MAC validating functions, BASECOMP relies on the fact that mobile networks can authenticate messages using multiple algorithms, including ZUC and SNOW3G [4,5]. To discover functions that implement these algorithms, we leverage a standard technique for cryptographic function identification [10,22,25]; we use magic constants (e.g., S-Box) to identify these functions. In Figure 5, the nodes written in ZUC and SNOW 3G indicate these functions. For Mediatek firmware, we utilize debug symbols to identify cryptographic functions as its encryption is processed by a custom hardware feature.

The next step is to identify a MAC validating function. The MAC validating function is probably one of the common ancestors of these cryptographic functions; however, it is unclear which is the right one. Instead of making a hasty conclusion, we adopt probabilistic inference. In particular, we create a random variable with initial probability[2] if a function is a common ancestor of the cryptographic functions. For Figure 5, we create random variables for functions f_1 and f_2 as they call both a ZUC function and a SNOW 3G function identified previously. Then, we introduce a preferable relationship (see §2) to prioritize lower common ancestors. This is based on our intuition that the MAC function should be located close to these cryptographic functions. Continuing with the example in Figure 5, we prefer f_1 over f_2 as f_2 calls f_1. Therefore, a preferable relationship is added between the two nodes. BASECOMP constructs a call graph of the firmware, iterates over each caller-callee pair, and registers the preferable relationship for common ancestors in the factor graph.

Step 2: Identifying the message type comparing function. Second, BASECOMP detects the message type comparing logic. Remember that the specification (the subclause 4.4.4.2 of TS 24.301 [3]) states that the integrity function must compare message types to allow exceptional messages before security exchange (i.e., in the insecure state). Unfortunately, it is very challenging to identify this logic in a deterministic way. Approaching this problem naively, we try locating a function that compares message types in the specification. However, without an expensive analysis (e.g., symbolic analysis), it is difficult to retrieve message types completely considering the

[2] §7.2 describes how the values of initial probability and preference relationships are decided.

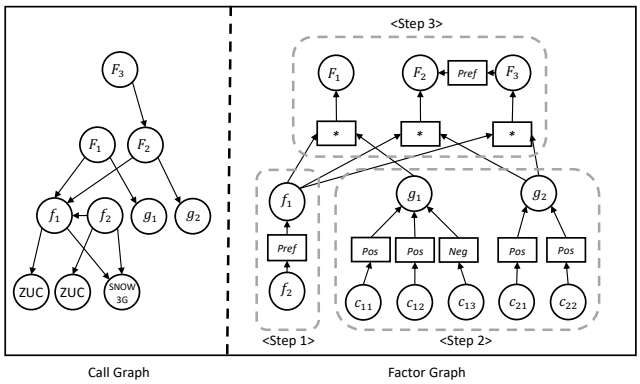


Figure 5: Example of a factor graph constructed.

complexity of modern optimization. More seriously, we cannot assume that the function is correct (which is actually not), as it might compare more or fewer message types than the specification defines.

To tackle this issue, BASECOMP represents the message type comparing logic also with probabilistic inference. In particular, BASECOMP iterates over every function in the firmware and collects constants for comparison. This analysis is syntactic, which makes it efficient but error-prone. Thus, we use probabilistic inference. If a certain constant is matched with an expected message type, we add a positive relationship to a random variable for that function. Otherwise, we add a negative relationship. Then, we compute each function's likelihood for the message comparing function. Note that this approach is more resilient to errors in analysis and buggy implementations because we do not draw any hasty conclusions. In Figure 5, g_1 and g_2 are functions that have comparisons with constants. For g_1, c_{11} and c_{12} creates a positive relationship to g_1 as they are expected message types, while c_{13} creates a negative one as it is not an expected message type. 2 positive relationships are added to g_2 as both c_{21} and c_{22} are expected message types.

Step 3: Putting it all together. Finally, we determine the integrity function using the same method in step 1. In more detail, we first identify common ancestors of the MAC function and message type comparing function. Just as in step 1, we then create a preferable relationship between the caller and the callee to prioritize lower common ancestors. In Figure 5, functions F_1, F_2 and F_3 are identified first and the preference relationship between F_2 and F_3 is created afterwards. Then, we compute the marginal probability of each function with the belief propagation algorithm. We can find the actual integrity function by verifying functions starting at the top of the rank.

5.2 Gathering Information from Firmware for Symbolic Analysis

Next, we need to acquire information about the baseband firmware's integrity protection function for BASECOMP's symbolic analysis. In this stage, we rely on human experts to examine the integrity protection function and write files like Figure 6. We leave automation for this part as future work because 1) cross-vendor implementations vary greatly from one another, 2) bugs in implementation impede deterministic reasoning, and 3) the complexity of analyzing implementations without any domain knowledge. For example, we discovered that a security state in Samsung baseband firmware is indistinguishable from other variables because it has almost no impact on authentication due to bugs (see §9 for more details). Human analysts can identify it as a mis-implementation (e.g., from debugging messages); however, automated methods are difficult due to their mere impact.

For its further symbolic analysis, BASECOMP requires a firmware-specific configuration file and a vendor-specific analysis module. The firmware-specific configuration (e.g., Figure 6a) specifies binary-related information, such as addresses of the integrity protection function, MAC validation function, and security state. It may also contain a deny-list of functions that should be skipped to avoid path explosion; the integrity

```
analysis:           ./analysis_samsung.py

# Functions for analysis
integrity_func:         0x4150AECD
mac_validation_func:    0x4150A3D6
security_state:         0x429B27C4

# Functions to skip to avoid path explosion
skip_funcs:
    - 0x40CECC87
    - 0x4057F5FB
```

(a) A firmware-specific configuration file of Galaxy S10 5G.

```
def symbolize(s, config):
  # Symbolizes a message buffer and a state variable
  msg_buf = s.solver.BVS('message_buffer', 32)
  s.regs.r0 = msg_buf

  sec_state = s.solver.BVS('security_state', 8)
  s.memory.store(config.security_state, sec_state)


def accepting(s, config):
  # Check if this return represents accepting a message
  return s.ret_val == 1
```

(b) A vendor-specific analysis module for Samsung.

```
def symbolize(s, config):
    struct = s.solver.BVS('struct', 32)
    s.regs.r1 = struct

    msg_buf = s.solver.BVS('message_buffer', 32)
    s.memory.store(struct + 4, msg_buf)

    def symbolize_security_state(s):
        return s.solver.BVS('security_state', 8)

    hook(config.security_state_func, symbolize_security_state)

...
```

(c) A vendor-specific analysis module for Mediatek.

Figure 6: Firmware-specific and vendor-specific files for BASECOMP's symbolic analysis

protection function is not as simple as its specification and contains other relevant features (e.g., replay protection or message handling). The vendor-specific analysis module specifies routines for analyzing a certain vendor's firmware and manages vendor-specific differences. For example, `symbolize` defines how to symbolize variables in the firmware. In particular, to analyze Samsung's firmware (Figure 6b), we need to symbolize a message buffer, which is passed as the first argument (r0 in ARM) of the integrity protection function (Line 3–4 in Figure 6b). Moreover, we need to symbolize a security state that is maintained as a global variable (Line 6–7). This module also defines how to determine whether a certain message is accepted or not (Line 10–12). For example, in Samsung, the integrity protection function returns 1 if the message can be accepted. This global variable's address may vary across different firmware versions. In order to determine this address, the analysis module uses the firmware-specific configuration (Line 7). Unlike Samsung, Mediatek firmware stores its message buffer as a structure (Line 2–6 in Figure 6c) and uses a function to indicate a security state (Line 8–11).

We wish to emphasize that thanks to our probabilistic inference, human experts only need to analyze 1–3 functions among 80K functions in average (see §7). Additionally, the vendor-specific analysis can also be applied to other firmware binaries from the same vendor thanks to their similarity. Defining variables in the firmware-specific configuration is sufficient to apply the analysis to various models. Thus, an analyst who uses BASECOMP only needs to write a few lines of code to support other firmware or vendors (see Table A2). Then, BASECOMP uses this information in the subsequent phase to analyze the integrity protection function automatically using symbolic analysis.

5.3 Symbolically Analyze Integrity Protection Following Specification

Building a reference model. To analyze integrity protection, BASECOMP requires a reference model for integrity protection. Luckily, this model is relatively simple unlike its implementation; therefore, we build it manually based on the subclause 4.4.4.2 of TS 24.301 [3]. Recall that this subclause defines plaintext messages that can be accepted without integrity protection. Particularly, we represent this model with symbolic constraints to compare them with those from the implementation. Table 2 shows constraints for plaintext messages based on the specification. It is worth noting that we must consider implicit constraints from the subclause, such as the security state or security header type to analyze the baseband firmware. In summary, based on the specification, the baseband firmware should accept only a specific set of messages in plaintext before exchanging the security context (i.e., only in the `INSECURE` state).

Building an implementation model using symbolic execution. As illustrated in Figure 7, BASECOMP analyzes the integrity protection via symbolic execution. BASECOMP analyzes only the integrity protection function without dealing with the entire baseband software following the concept of under-constrained symbolic execution [50] (Line 2–5). That is, BASECOMP runs symbolic execution from the beginning of the function to its return. BASECOMP marks the variables of interest to track, such as the message buffer and security state variable before running symbolic execution (Line 4). As a result, BASECOMP evaluates how those symbolic variables are used in the integrity protection function. Specifically, BASECOMP collects symbolic variables and constraints associated with an accepted message (Line 25–29) without performing MAC validation (Line 16–17). Such a message is particularly interesting because it can be crafted by an attacker with no valid key. Additionally, BASECOMP avoids path explosion by skipping manually provided irrelevant functions, such as those for logging (Line 19–21).

```
1  def symbolically_analyze_integrity_func(config):
2    # Make an under-constrained state for symbolic analysis
3    states = [
4      config.analysis.symbolize(State(config.integrity_func))
5    ]
6    constraints = []
7
8    while states:
9      cur_state = states.pop()
10     while True:
11       instr = cur_state.next_instr()
12
13       if instr.type == CALL:
14         target_func = instr.operand
15         # Drop a path that validates MAC (i.e., not plaintext)
16         if target_func == config.mac_validation_func:
17           break
18         # Skip irrelevant functions to avoid path explosion.
19         elif target_func in config.skip_funcs:
20           cur_state.skip_instr()
21           continue
22
23       # If the integrity function accepts a current message, add
24       # the path constraints for return.
25       elif instr.type == RET
26         and instr.operand == config.integrity_func:
27         if config.analysis.accepting(cur_state):
28           constraints.append(cur_state.constraints)
29         break # Finish analyzing this path.
30
31       # If a branch is symbolic, fork states for exhaustive
32       # analysis and interally update constraints.
33       elif cur_state.is_symbolic_branch(instr):
34         state_forked = cur_state.fork(instr)
35         if state_forked:
36           states.push_back(state_forked)
37
38       cur_state.exec_instr(instr)
39
40   return constraints # Returns constraints for plaintexts
```

Figure 7: Pseudocode of symbolic analysis for integrity protection.

Comparative analysis. After that, BASECOMP identifies a list of acceptable message types and additional conditions (e.g., message types and security header types) by concretizing symbolic constraints for the state variable. Then, BASECOMP compares them with our model from the specification to discover inconsistencies. BASECOMP will report any mismatches, and

Table 2: Constraints of plaintext NAS messages that can be accepted without security protection (in the `INSECURE` state).

SECURITY State	Security Header Type	Message Type	Other Conditions
INSECURE	0 (Not Protected)	Identity Request	Identity Type is IMSI
INSECURE	0 (Not Protected)	Authentication Request	
INSECURE	0 (Not Protected)	Detach Accept	
INSECURE	0 (Not Protected)	Authentication Reject	
INSECURE	0 (Not Protected)	Attach Reject	EMM Cause ! = 25
INSECURE	0 (Not Protected)	Tracking Area Update Reject	EMM Cause ! = 25
INSECURE	0 (Not Protected)	Service Reject	EMM Cause ! = 25

Table 3: Components and lines of code (LoC) of BASECOMP.

Component	LoC (Python)
Loading firmware to IDA	1,798 lines
Probabilistic inference	468 lines
Symbolic analysis of integrity protection	529 lines
MIPS16e2 support	1,957 lines
Total	4,752 lines

we need to further analyze their implication to determine whether they can lead to security-critical issues. We applied this approach to Samsung, MediaTek, and srsRAN baseband and discovered 29 bugs that can be categorized into 15 types (E1–E15), including one NAS AKA bypass vulnerability in Samsung (E4).

BASECOMP determines message acceptance conservatively to avoid false positives. More specifically, BASECOMP concludes that a message is rejected if the same message (i.e., with the same headers and payload) can be both accepted or rejected according to some external variable. This can happen if the firmware implements emergency or debugging features. In fact, Samsung's integrity protection function allows every message as plaintext if an emergency call is ongoing. Using this method, we can avoid such cases automatically.

6 Implementation

We implemented our prototype, BASECOMP, mainly in Python with 4.7k lines of code as shown in Table 3. To analyze baseband software, we used the state-of-the-art binary analysis tool, IDA Pro v7.6 [26]. It provides useful Python APIs for binary analysis, ranging from a basic disassembler to even a remarkable decompiler called Hex-Rays. In addition, IDA Pro supports static analysis on top of the decompiled source code from Hex-Rays. Thus, we utilized this feature for analyzing integrity protection (§5). To load firmware to IDA, we followed BaseSpec's instructions [33]. The probabilistic inference engine is built on pgmpy [6], a python library for probabilistic graphical models. To build the call graph for firmware, we relied on NetworKit [58], a large-scale network analysis tool. Moreover, we utilized *angr*, a promising binary analysis framework [57], to apply symbolic execution.

Table 4: The rank and probability of the integrity checking function for each firmware by the value of p.

Firmware	Size (KB)	# of functions	p=0.2		p=0.3		p=0.4	
			Rank	Prob	Rank	Prob	Rank	Prob
G950	42212	64184	1	0.941	1	0.823	1	0.582
G955	42791	61596	1	0.941	1	0.823	1	0.582
G960	42542	74311	1	0.939	1	0.811	1	0.561
G965	42591	74259	1	0.939	1	0.811	1	0.561
G970	45105	91656	1	0.992	1	0.942	1	0.716
G975	45349	75299	1	0.945	1	0.816	1	0.562
G977	45409	92416	1	0.992	1	0.942	1	0.715
G991	68189	103334	3	0.702	3	0.607	3	0.590
G996	67902	107536	1	0.779	1	0.627	1	0.440
G998	67901	103117	3	0.703	3	0.615	3	0.551
Pro 7	18193	48350	2	0.999	2	0.996	2	0.933
A31	23004	93550	2	0.999	2	0.998	2	0.933
A31 (Latest)	23036	93754	2	0.999	2	0.999	2	0.968
A03s	17212	64942	2	0.999	2	0.999	2	0.968
A145	17372	65075	2	0.999	2	0.999	2	0.968
srsran	95083	95842	1	0.799	1	0.690	1	0.529

7 Evaluation

In this section, we evaluate our approach, BASECOMP, to answer the following questions:

- How effectively can BASECOMP find the integrity protection function in the firmware? (§7.2)
- How effectively can BASECOMP discover bugs in message authentication? (§7.3)
- How effective is BASECOMP in finding bugs in integrity protection, compared to existing dynamic techniques? (§7.4)
- How long does it take to run BASECOMP? (§7.5)

7.1 Evaluation Setup

Firmware. We collected a total of 16 images, as shown in Table A1. We first downloaded 10 firmware images for smartphones with Samsung's baseband (Galaxy S8, S9, S10, and S21 series) from Samsung's cloud server for firmware updates. In addition, to analyze various vendors' implementations, we obtained 2 from MediaTek and compiled the open-source project, srsRAN. For MediaTek, we found one for MEIZU Pro 7 on the web and the other one for Galaxy A31 in Samsung's cloud server, respectively. We wish to emphasize that the firmware for Galaxy A31 is based on MIPS16e2 with application-specific extensions, and srsRAN is compiled into x86. This demonstrates BASECOMP's effectiveness in supporting diverse architectures.

Machine. We performed all the following experiments on Windows 11 Pro equipped with AMD Ryzen 9 5900X 12-Core Processor, 3.70GHz, 64GB DDR4 RAM.

7.2 Identifying Integrity Protection

Effectiveness. For its probabilistic inference, BASECOMP needs to define parameters for functions in a factor graph, namely p_{Pos}, p_{Neg}, and p_{Pref}. Currently, we arbitrarily choose these values based on parameter p as follows.

$$p_{Pos} = p$$
$$p_{Neg} = 0.5 + (0.5 - p)/2$$
$$p_{Pref} = p$$

We intentionally lower the effect of p_{Neg} than p_{Pos} as we prefer a function with one matching comparison and one non-matching comparison over a function with no comparisons.

To demonstrate its effectiveness, we run BASECOMP's probabilistic inference to all firmware binaries in Table A1 with the default setting ($p = 0.2$). As shown in Table 4, BASECOMP can discover the integrity protection for all firmware within the top three ranks. Taking into account the size and the number of functions in these firmware binaries, we believe that BASECOMP's probabilistic inference significantly reduces the amount of manual effort required by users; baseband firmware is extremely complicated, including more than 80K functions on average.

We observed that BASECOMP's probabilistic inference is effective across models and vendors. For example, BASECOMP is able to successfully detect the integrity protection function for S950 and S996 in the top rank even though their probabilities (0.941 and 0.779, respectively) indicate their drastically different shapes. Moreover, BASECOMP discovers the integrity protection function of MediaTek's MEIZU Pro 7 in the second rank. This implies that our technique is not vendor-specific and can be utilized generally.

Reasons for not being ranked at the top. BASECOMP fails to locate the genuine integrity protection function at the first rank for two reasons. First, as BASECOMP favors lower common ancestors, sub-routines of the actual integrity protection function can be placed in higher ranks because they can also contain both the MAC validating and message comparing logic. For example, we concluded that the top-ranked function of Pro 7 is such an example. Second, it is possible that firmware re-implements integrity protection for other purposes. For example, we found that G998 contains additional integrity protection that seems to be used for testing [32]. This is not the integrity protection function for security that we seek; however, BASECOMP is unable to distinguish the two due to its lack of understanding of their usages.

Robustness. We also evaluate if BASECOMP's probabilistic inference is sensitive to the parameter p. For that, we repeat the previous evaluation with p's value to 0.3 and 0.4. As shown in Table 4, BASECOMP is resilient to changes in p. Even though the final probabilities are different, none of the ranks are affected by the value of p. This demonstrates that our intuitive but seemly arbitrary selection of functions in a factor graph is acceptable.

7.3 Analyzing Integrity Protection

Table 5 illustrates BASECOMP's comparative analysis results for integrity protection. We identified acceptable plaintext messages and their constraints in detail via symbolic execution. BASECOMP discovers mismatches by comparing these results with constraints from the specification (Table 2). In summary, BASECOMP reports a total of 34 mismatches and 29 of them were actual bugs. We identified actual bugs by testing them over-the-air. The remaining 5 were all false positives where the integrity protection was applied outside of our analysis scope in an ad-hoc manner. We describe the details of the false positives at the end of this section.

The mismatches are classified by the constraints we extracted from symbolic execution. For instance, the third row of Table 5 shows that the integrity protection function accepts an `Identity Request` message as plaintext (i.e., security header type is 0 that represents `Not Protected`) in the `INSECURE` state and when the Identity Type field in the message payload is not IMSI. It is worth noting that because baseband software only offers integer values for messages, we identified message types and fields from the specification for clarity.

We categorized the bugs into 15 types (E1–E15). Each bug type is described with an analysis of its root cause and the security implication will be further discussed in the following section (§8).

Root cause analysis (E1–E3). Subclause 4.4.4.2. in TS 24.301 specifies messages that can be accepted as plaintext. In addition to the message type, there are additional conditions for certain messages. Bugs E1–E3 result from the absence of these additional checks. In particular, E1–E3 are caused by allowing the EMM Cause of the `Attach Reject`, `Tracking Area Update Reject` and `Service Reject` to be #25, which is not allowed by the specification.

Root cause analysis (E4). E4 (only in Samsung) allows *any* plaintext message to be accepted before security activation if its security header type is invalid (i.e., neither 0, 1, 2, 3 nor 12). This allows an attacker to bypass NAS AKA, becoming possible any malicious behavior such as SMS phishing. This is caused by improper validation of an incoming message.

Figure 8 illustrates the simplified code for integrity protection in Samsung. The `CheckHeader` function in Figure 8 verifies the security header type in the `INSECURE` state (Line 43 – 49). Unfortunately, the function only compares the security header type with zero and always returns `true` if it is not. Notably, other checks (i.e., `CheckSeq` and `ValidMac`) are invoked only when the message's security header type is either 1, 2, or 3. In addition, there is a dedicated routine for when the security header type is 12; used for `Service Request` messages. As a result, if the security header type of an incoming message is larger than 3 and not 12, the message will be blindly accepted even though it is not allowed according to the specification.

Root cause analysis (E5–E11). Bugs E5–E11 are related to incorrect handling of plaintext messages in the `SECURE` state. In particular, the specifications only define a few types of messages that can be accepted as plaintext before security activation (*i.e.*, in the `INSECURE` state). After security activation,

Table 5: The list of plaintext messages that satisfies the firmware's integrity protection function and their condition but mismatches the specification (Table 2). Cells are marked with circles if the vendor accepts the specific mismatch. Cells marked FP are false positives and further described in §7.3.

SECURITY State	Security Header Type	Message Type	Other Conditions	Mismatches in Samsung	Mismatches in MediaTek	Mismatches in srsRAN	Errors	Implication
INSECURE	3	Secure Mode Command		FP				
INSECURE	0 (Not Protected)	Identity Request	Identity Type ! = IMSI			FP		Info leak [29, 43, 48]
INSECURE	0 (Not Protected)	Attach Reject	EMM Cause == 25		FP	●	E1	DoS [48]
INSECURE	0 (Not Protected)	Tracking Area Update Reject	EMM Cause == 25		FP	●†	E2	DoS [12, 13, 65]
INSECURE	0 (Not Protected)	Service Reject	EMM Cause == 25		FP	●	E3	DoS [13]
INSECURE	! = 0, 1, 2, 3, 12	*		●			E4	Auth bypass
SECURE	0 (Not Protected)	Identity Request	Identity Type == IMSI	○	○	●	E5	Info leak [29, 43, 48]
SECURE	0 (Not Protected)	Authentication Request		○	○	●	E6	Location leak [29, 31]
SECURE	0 (Not Protected)	Detach Accept		◑	◑	●†	E7	-
SECURE	0 (Not Protected)	Authentication Reject		◑	◑	●	E8	DoS
SECURE	0 (Not Protected)	Attach Reject	EMM Cause ! = 25	◑	◑	●	E9	DoS [54]
SECURE	0 (Not Protected)	Tracking Area Update Reject	EMM Cause ! = 25	○	◑	●†	E10	DoS [13, 54, 65]
SECURE	0 (Not Protected)	Service Reject	EMM Cause ! = 25	◑	○	●	E11	DoS [13, 54]
*	0 (Not Protected)	Detach Request				●	E12	DoS [12, 28, 35]
*	0 (Not Protected)	EMM Information				●	E13	Info spoofing [35, 48, 49]
*	0 (Not Protected)	EMM Status				●	E14	- [35]
*	4	*				●	E15	Auth bypass
Total number of			Mismatches	9	10	15		
			Bugs	8	7	14		

●: New bugs (neither bug nor its root cause previously reported), ◑: Duplicated bugs (not previously reported, but bugs with identical root causes were), ○: Old bug (previously reported) †: This bug has no implication due to the absence of handlers in the current implementation.

the baseband software should not accept these messages as plaintext. However, all tested vendors allow these messages as plaintext even after security activation.

The `CheckHeader` function in Figure 8 checks whether the given message type is allowed in plaintext according to the specification (subclause 4.4.4.2. in TS 24.301). This should be checked only in the `INSECURE` state, as shown in Line 43–44. However, developers misunderstood the specification and implemented the baseband software to allow them even in the `SECURE` state (Line 34–37). As a result, seven messages are incorrectly accepted by Samsung and MediaTek's baseband (E5–E11) as the code of the integrity protection function for the two vendors is nearly identical. In the case of srsRAN, the implementation does not track the security state in the integrity protection function. This makes the `INSECURE` and `SECURE` states indistinguishable. Consequently, plaintext messages that should only be allowed in the `INSECURE` state are also allowed in the `SECURE` state (E5–E11).

Root cause analysis (E12–E14). Apart from the message types specified in Subclause 4.4.4.2. in TS 24.301, srsRAN 's implementation accepts 3 additional message types in plaintext: `Detach Request`, `EMM Information`, and `EMM Status` (E12–E14).

Root cause analysis (E15). Lastly, srsRAN accepts the security header type — integrity protected and ciphered with new EPS security context — without any integrity check (E15). Notably, according to the specification, this type should only be used for the `Security Mode Complete` message. As a result, this error leads to NAS AKA bypass, similar to E4.

False positives. Due to the limited scope of BASECOMP (*i.e.*, only the integrity protection function), we found that BASECOMP can result in false positives — mismatches in the integrity protection function that are not bugs. In particular, BASECOMP reports that the integrity protection function accepts the `Security Mode Command` message in the `INSECURE` state if security header type is 3, which stands for integrity protected with new EPS security context (see Table 1). This special security header type can only be used for `Security Mode Command` and should be integrity protected. However, after post-analysis, we discovered that the message's integrity is validated in another dedicated routine as `Security Mode Command` has a special role in integrity protection. Note that `Security Mode Command` is a message for establishing a security context for integrity protection. BASECOMP also reports that srsRAN accepts `Identity Request` messages in the `INSECURE` state without IMSI checking and MediaTek accepts `Attach Reject`, `Tracking Area Update Reject` and `Service Reject` messages in the `INSECURE` state without checking the EMM Cause. However, these cases are later validated in routines after the analyzed integrity protection function alike the previous case. This demonstrates a drawback of static analysis similar to BASECOMP. However, BASECOMP is useful as this can be complemented with over-the-air testing.

7.4 Comparison to Dynamic Testing

To compare BASECOMP with dynamic testing, we review `DoLTEst` [48] and `DIKEUE` [29] that use dynamic testing for analyzing integrity protection. For comparison, we re-run `DoLTEst`, and for `DIKEUE`, we referred to the results in the paper because only their FSM modules were open-sourced. Table 6 shows the integrity protection bugs that BASECOMP and other approaches discover for Samsung devices. This demonstrates that BASECOMP can cover more types of integrity protec-

```
// These are arbitrary named for better explanation.
enum SecState { SECURE, INSECURE };

// A state variable for a security context.
SecState sec_state;

bool IntegrityProtection(void* message) {
  // Returns true if the 'message' is valid to be accepted.
  if (CheckHeader(message)
      && (!IsProtected(message) || CheckSeq(message))
      && (!IsProtected(message) || ValidateMac(message)))
    return true;
  else
    return false;
}

bool IsProtected(void* message) {
  uint8_t sec_hdr_type = GetSecHdrType(message);
  return sec_hdr_type != 0 && sec_hdr_type <= 3;
}

bool CheckAllowableInNonSecure(void* message) {
  // Returns true if the 'message' is specified
  // as exceptions in TS 24.301.
  ...
}

bool CheckHeader(void* message) {
  uint8_t sec_hdr_type = GetSecHdrType(message);

  if (sec_state == SECURE)
  {
    if (sec_hdr_type == 0) {
      // BUG #1: In the SECURE state,
      // plaintext messages should not be accepted.
      return CheckAllowableInNonSecure(message)
    }
    else if (IsProtected(message))
      return true;
    else
      return false;
  } else { // INSECURE
    if (sec_hdr_type == 0)
      return CheckAllowableInNonSecure(message);
    else {
      // BUG #2: In the INSECURE state,
      // this function returns true
      // if sec_hdr_type is non-zero yet invalid.
      return true;
    }
  }
}
```

Figure 8: Simplified version of the decompiled code for integrity protection in Samsung.

tion bugs than recent dynamic approaches. This happens for two reasons. First, to tame an extremely large input space, dynamic approaches are limited in testing a few critical messages (such as `Identity Request` and `Authentication Request`), while leaving out others (E7–E11). Second, dynamic testing only focuses on semantically valid messages, similarly to reduce the search space. Unfortunately, to discover E4, a semantically invalid message with non-zero security header type but without Authentication Header must be created (see Figure 2). It is worth noting BASECOMP is *more restricted* than dynamic methods in terms of applicability (e.g., devices or other features to test). Nevertheless, BASECOMP can complement dynamic testing in terms of completeness for analyzing integrity protection.

Table 6: Integrity protection bugs in Samsung devices discovered by BASECOMP and recent dynamic approaches, `DoLTEst` and `DIKEUE`.

	E4	E5	E6	E7	E8	E9	E10	E11
BASECOMP	✓	✓	✓	✓	✓	✓	✓	✓
`DoLTEst` [48]		✓						
`DIKEUE` [29]		✓	✓					

Table 7: Elapsed time for BASECOMP's probabilistic inference and symbolic analysis, in seconds.

Firmware	Probabilistic Inference					Symbolic Analysis	Total
	Call Graph	Step 1	Step 2	Step 3	Total		
G950	15.7	13.6	538.3	13.2	580.9	473.2	1054.0
G955	14.4	13.7	471.1	12.6	511.8	474.8	986.6
G960	21.4	11.0	620.6	21.2	674.1	15.3	689.4
G965	21.1	12.2	563.3	22.8	619.4	17.9	637.3
G970	42.6	62.9	783.5	52.7	941.8	28.7	970.5
G975	21.8	6.1	617.6	13.3	658.8	17.3	676.0
G977	49.8	76.1	773.7	52.5	952.1	62.6	1014.7
G991	82.7	101.9	754.3	27.8	966.6	45.9	1012.6
G996	94.7	46.2	762.8	19.6	931.9	45.8	977.8
G998	80.2	951.9	718.8	63.5	1814.4	46.5	1861.0
Pro 7	10.1	-	294.0	17.0	312.1	11.1	332.2
A31	45.1	-	633.7	34.8	713.5	67.3	780.8
A31 (Latest)	49.2	-	631.5	77.9	758.6	57.9	816.5
A03s	15.4	-	421.2	118.7	555.2	6.3	561.5
A145	15.4	-	459.5	48.4	523.4	57.7	581.1
srsran	47.2	308.5	405.0	18.4	779.2	30.0	809.2
Average	39.2	145.8	590.6	38.4	768.9	91.2	860.1

7.5 Performance Evaluation

Table 7 shows the elapsed time of BASECOMP, including call graph construction (Call graph), each step of probabilistic inference (Step 1, Step 2, and Step 3), and symbolic analysis (Symbolic Analysis). To determine the elapsed time, we run BASECOMP for each firmware three times and average their results. In short, BASECOMP requires 860.1 seconds on average. Step 2 in probabilistic inference is the most time-consuming because the belief propagation algorithm is expensive for a large factor graph; every function with a comparison to an expected value will become a node in the graph in Step 2. BASECOMP's symbolic analysis is quite fast (91.2 seconds on average) thanks to its firmware-specific configuration, which allows BASECOMP to avoid functions that cause path explosion. We found that angr fails with out of memory without such information.

8 Security Analysis and Case Study

In this section, we further discuss the implications of the bugs that we discovered (E1–E15). We built our environment using universal software radio peripheral (USRP) with srsRAN [21] and validated all of our findings over-the-air. Note that we used Galaxy S10e and Galaxy S10 5G, Galaxy A31, and srsUE as representative devices for testing Samsung, MediaTek, and srsRAN basebands, respectively.

8.1 NAS AKA Bypass Vulnerability (E4)

We discovered that E4 in Samsung can cause a critical issue, leading to LTE NAS authentication bypass. Using this vulnerability, attackers with a malicious base station can bypass NAS authentication and send arbitrary NAS messages in plaintext. Although we demonstrated this vulnerability on Galaxy S10 5G, we believe that it will affect most smartphones from Samsung.

Exploit details. We further describe our proof-of-concept exploits to highlight the severity of this bug. As mentioned earlier, BASECOMP discovered that any plaintext NAS message can bypass integrity protection if its security header type is something other than 0, 1, 2, 3, and 12. Using this flaw, an attacker can send any prohibited messages such as `Identity Request` (with IMEI as identity type), `EMM Information`, and even `Attach Accept` in plaintext, prior exchanging security context (i.e., in `INSECURE` state).

We demonstrate this attack by delivering an arbitrary SMS with a malicious base station[3], as shown in Figure 10. Figure 9 illustrates our exploitation steps. First, when a smartphone sends an `Attach Request` message to connect to the malicious base station, the attacker responds with a plain `Attach Accept` message without key agreement procedures, including `Security Mode Command/Complete`. This violates the specification because the `Attach Accept` message should be integrity protected. However, due to the vulnerability, the baseband software accepts this plaintext message if the security header type is larger than 3 and not 12. After receiving the `Attach Complete` message from the smartphone, the attacker sends a `Downlink NAS Transport` message with SMS as plaintext by abusing the vulnerability again. In this message, the attacker can add any SMS data (*e.g.*, the phone number of the sender, the received time, and even the SMS content), as shown on the right side of Figure 10. We believe this exploit can lead to further attacks like SMS phishing [8].

Notably, this is only an illustrative implication of this vulnerability. Simply speaking, this vulnerability allows us to *severely break* the security of mobile communication in smartphones with Samsung baseband. For instance, the attacker can gather IMEI by sending the `Identity Request` message, which is only allowed in the `SECURE` state according to the specification. Furthermore, the attacker can modify the current time on smartphones using the `EMM Information` message. We responsibly disclosed the vulnerability details and possible attack scenarios to the manufacturer, Samsung. In response, Samsung assigned the vulnerability *critical* severity and awarded us with a bug bounty.

8.2 Mishandling of Plain Messages (E5 – E11)

We discovered that the integrity protection error cases, E5 through E11, may cause security issues such as denial of service [54] and information leakage [15]. For instance, attackers may inject reject messages, such as an `Attach Reject` message, to forcibly release the victim's connection. To exploit these errors, the attacker can leverage a man-in-the-middle attack [52,53] or a signal injection attack [17,63] to inject malicious messages into the communication between smartphones and the base station. However, because these error cases occur after the security context exchange (i.e., in the `SECURE` state), the RRC layer would be already secured. Therefore, attackers would have to inject messages before the victim activates the RRC layer security. Note that a man-in-the-middle attacker can easily exploit this timing, as the attacker can control the protocol message flow and not activate the RRC layer security. Alternatively, attackers can leverage other vulnerabilities to incapacitate RRC layer security to inject seven types of NAS messages in plaintext by exploiting E5–E11. Also, we responsively disclosed our findings to the corresponding vendors.

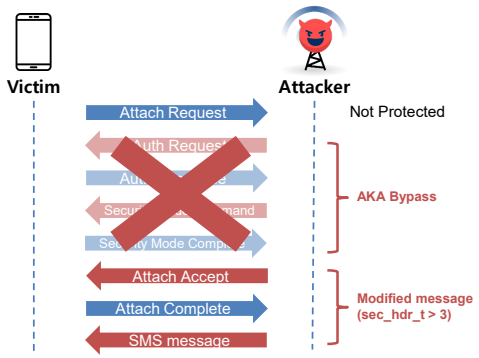


Figure 9: AKA bypass vulnerability and SMS-phishing attack

8.3 Security analysis for srsRAN

Using BASECOMP, we found that attackers can exploit all bugs in srsRAN except for those where the corresponding handlers are not implemented (E2, E7, and E10) or the Samsung-specific bug (E4). In this section, we discuss srsRAN-specific bugs (E12–15).

We found srsRAN accepts other message types that are not allowed in the specification (E12–14). These issues result in diverse implications. In particular, attackers can change the internal state of a UE and deactivate the EPS bearer, which is essential for the UE to use the internet, by exploiting E12. Also, attackers may exploit E13 to inject an `EMM Information` message and manipulate the network time and network name. We verified that E14 has no implication because it is only used to report error conditions. As a result, no implication for a UE accepting a plain `EMM Status` message, which is accepted by E14, has yet been reported.

Lastly, we discovered that E15 in srsRAN can also cause a NAS AKA bypass, which has the same implication as E4. Compared to E4, E15 has a wider attack surface in terms of security state; attackers can exploit E15 regardless of the

[3]The demo video is uploaded on https://youtu.be/4yM3uyiRzvo

Figure 10: Exploit environment (left side) to demonstrate the NAS AKA bypass vulnerability (E4), delivering an arbitrary SMS message (right side).

existence of the security context in srsRAN because srsRAN does not track the security state in the integrity protection function. We demonstrated all the bugs on srsUE (release 22.04) and responsibly disclosed all the bugs.

9 Discussion & Limitations

Challenges in full automation. BASECOMP is currently a semi-automated system that involves manual analysis. We've attempted to fully automate this system but decided to leave it as future work due to several challenges. First, there are numerous ways to implement even a single specification. For example, Samsung's implementation transfers a message buffer to its integrity protection function as a simple array, whereas MediaTek employs a complicated structure. Moreover, Samsung and MediaTek implement a security state via a global variable and a function, respectively. As a result, we cannot assume any prior knowledge for their implementation; therefore, we rely on humans to manage such diversity. Second, the incorrect implementation of baseband firmware significantly hinders automated analysis. For instance, to analyze integrity protection, we need to identify the `SECURITY` state. According to the specification, certain messages should be processed only in the `SECURE` state but not in the `INSECURE` state. For example, if an `Attach Accept` message is integrity protected with the security header type 1, the firmware should validate its MAC only in the `SECURE` state; in the `INSECURE` state, this message should be rejected regardless of its MAC. However, due to a bug in Samsung, this message is always MAC validated regardless of the `SECURITY` state, making it difficult for the `SECURITY` state to be distinguished. Finally, the integrity protection function is not as simple as its specification. In reality, the functionality is not isolated but includes extra security checks (e.g., replay protection), debugging features, and message-handling routines. Thus, these parts significantly complicate fully automated analysis.

Limitations. Despite its success, BASECOMP also has several limitations. First, BaseComp is built with substantial manual efforts. However, the majority of its modules are reusable, thereby significantly reducing future efforts. Largely, BaseComp's modules can be categorized into three: specification-driven, vendor-specific, and firmware-specific modules. Specification-driven modules such as the reference model for integrity protection are compatible with all firmware. Thus, analysts are no longer required to care about this. Although vendor-specific modules should be implemented per vendor, it is highly reusable due to the limited number of baseband vendors. Furthermore, we have already implemented those for Samsung and MediaTek, two of the top three baseband vendors. Finally, we need to obtain firmware-specific information (Figure 6a) for testing a new image. We believe that this is not challenging; according to our experience, it took only a few minutes thanks to code reuse within the same vendor.

Second, we fail to support the Qualcomm baseband because of its `Hexagon` architecture. Unfortunately, the state-of-the-art tools for static analysis (*e.g.*, IDA Pro) that BASECOMP relies on do not support `Hexagon` due to its extraordinary design. Notably, `Hexagon` incorporates Very Long Instruction Word (VLIW), embedding instruction-level parallelism (ILP) in its instruction set architecture, complicating static analysis. Due to similar reasons, recent work for analyzing baseband software [24,33,41] is also limited to Samsung and MediaTek.

Third, BASECOMP has a limited scope of analysis (*i.e.*, integrity protection). For instance, BASECOMP does not investigate the correctness of other message handlers because it is extremely challenging to compare them with the specification due to their diversity and complexity. Even though several problems may exist in handlers, as we have observed in many dynamic approaches [29,48], BASECOMP can only support integrity protection. This is a fundamental limitation of symbolic analysis; however, we believe that this approach is still a valuable approach that we need to explore further for baseband security (§7.4).

10 Related work

Blackbox analysis for baseband firmware. There have been several studies on analyzing software bugs or vulnerabilities of cellular protocols implemented in baseband firmware. In the early stage, researchers performed blackbox analysis on a cellular device without analyzing baseband firmware directly. For this, they built a physical testing environment using open-source cellular projects [9,21,46,62] and software-defined radios [18,47]. Then, they sent crafted messages to a target device in order to analyze private information leakage [54,55], network downgrading [39,51], and memory-related vulnerabilities in SMS or cell broadcast messages [44,45,59]. Recent approaches have further systematized the methodology to generate abnormal messages [19,29,31,35,48,52]. Several other studies have adopted a similar approach on various layers, protocols, or domains in cellular networks, such as

VoLTE [34, 37], SS7/Diameter [27], uplink messages [14, 35], or lower layers [38, 53, 63]. However, these approaches have limited applicability as they require a physical testing environment and devices. To remedy this issue, Maier *et al*. [41] and Hernandez *et al*. [24] recently proposed an emulation-based approach. In particular, they manually analyzed baseband firmware to emulate it and then ran a fuzzer, such as AFL++ [42] in order to uncover vulnerabilities. While these approaches are advantageous as they do not require comprehensive understanding of baseband firmware, they are highly likely to miss potential, critical bugs or vulnerabilities deeply hidden in the firmware (as discussed in §9).

Whitebox analysis for baseband firmware. On the other hand, several studies have directly analyzed baseband firmware [11, 20, 60]. Due to the high complexity of baseband software, many studies have relied on manual analysis. In particular, Weinmann [60] utilized the JTAG debug interface to analyze memory-related bugs of GSM protocol stacks in baseband firmware. Golde *et al*. [20] and Cama [11] analyzed Exynos baseband firmware using memory dumps, discovering RCE 0-days. While these approaches provide promising insights into baseband analysis, they are limited by the requirement of physical memory dumps, which is not supported in recent devices. Firmalice [56] suggests a generic system for backdoor detection. However, it expects a hard-coded credential in relatively simple binaries compared to baseband firmware. Due to the complexity and size of baseband firmware, we need a specialized system like BASECOMP to support it. Similar to our work, a recent study, `BaseSpec` [33], has proposed a technique to compare implementation with the specification relying on manual and comparative analysis. Unlike `BaseSpec`, BASECOMP focuses on integrity protection, which requires semantic reasoning with cryptography. Moreover, BASECOMP employs probabilistic inference to reduce the amount of manual effort significantly. As a result,BASECOMP can discover logical inconsistencies in integrity protection, resulting in the discovery of critical security vulnerabilities.

11 Conclusion

In this paper, we propose BASECOMP, a static approach to analyze the integrity protection of baseband software. To this end, we use a hybrid approach combining probabilistic inference and comparative analysis; after locating the implementation of integrity protection, we compare it with the specification to identify inconsistencies. As a result, we discovered several mismatches between the specification and implementations from Samsung and MediaTek, and a total of 29 bugs. Thanks to its comprehensive analysis, BASECOMP successfully discovered new vulnerabilities in baseband that existing blackbox approaches had missed including the NAS authentication bypass, a critical vulnerability in Samsung baseband.

12 Acknowledgment

We thank the shephered and the anonymous reviewers for their helpful comments and suggestions. This work was supported by Korea-U.S. Joint Research Support Program funded by the Ministryof Science and ICT through the National Research Foundation of Korea (2022K1A3A1A9109426711) and by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01202, Regional strategic industry convergence security core talent training business).

References

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

[2] 3GPP. TS 24.007; Mobile radio interface signalling layer 3; General aspects, 2022.

[3] 3GPP. TS 24.301; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3, 2022.

[4] 3GPP. TS 35.216; Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2; Document 2: SNOW 3G specification, 2022.

[5] 3GPP. TS 35.222; Specification of the 3GPP Confidentiality and Integrity Algorithms EEA3 & EIA3; Document 2: ZUC specification, 2022.

[6] Ankur Ankan and Abinash Panda. pgmpy: Probabilistic graphical models using python. In *Proceedings of the 14th Python in Science Conference (SCIPY 2015)*. Citeseer, 2015.

[7] David Berard and Vincent Fargues. How to design a baseband debugger. In *Information and Communication Technology Security Symposium (SSTIC)*, Rennes, France, jun 2020.

[8] Hugo Bijmans, Tim Booij, Anneke Schwedersky, Aria Nedgabat, and Rolf van Wegberg. Catching phishers by their bait: Investigating the dutch phishing landscape through phishing kit detection. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, August 2021.

[9] David A Burgess and Harvind S Samra. The OpenBTS Project. *Open Source Cellular Infrastructure*, 2008.

[10] Joan Calvet, José M Fernandez, and Jean-Yves Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[11] Amat Cama. A walk with Shannon. In *OPCDE*, 2018.

[12] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Dongfang Zhao. Seeing the forest for the trees: Understanding security hazards in the {3GPP} ecosystem through intelligent analysis on change requests. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Boston, MA, August 2022.

[13] Yi Chen, Yepeng Yao, XiaoFeng Wang, Dandan Xu, Chang Yue, Xiaozhong Liu, Kai Chen, Haixu Tang, and Baoxu Liu. Bookworm game: Automatic discovery of lte vulnerabilities through documentation analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2021.

[14] Merlin Chlosta, David Rupprecht, Thorsten Holz, and Christina 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.

[15] Adrian Dabrowski, Nicola Pianta, Thomas Klepp, Martin Mulazzani, and Edgar Weippl. IMSI-Catch Me If You Can: IMSI-Catcher-Catchers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[16] Guillaume Delugré. Reverse engineering a Qualcomm baseband. In *28th Chaos Communication Congress*, Berlin, Germany, dec 2011.

[17] Simon Erni, Patrick Leu, Martin Kotuliak, Marc Röschlin, and Srdjan Čapkun. Adaptover: Adaptive overshadowing of lte signals. *arXiv preprint arXiv:2106.05039*, 2021.

[18] USRP Ettus. B210.

[19] Kaiming Fang and Guanhua 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, September 2018.

[20] Nico Golde and Daniel Komaromy. Breaking Band: reverse engineering and exploiting the shannon baseband. *REcon*, 2016.

[21] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D Sutton, Pablo Serrano, Cristina Cano, and Doug 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.

[22] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *International Workshop on Recent Advances in Intrusion Detection*, pages 41–60. Springer, 2011.

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

[24] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. FirmWire: Transparent dynamic analysis for cellular baseband firmware. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2022.

[25] Hex Rays. Findcrypt. https://hex-rays.com/blog/findcrypt/, 2006.

[26] SA Hex-Rays. IDA: Hex-Rays. https://www.hex-rays.com/products/ida.

[27] Silke Holtmanns, Siddharth Prakash Rao, and Ian 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.

[28] Syed Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa 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, February 2018.

[29] Syed Hussain, Imtiaz Karim, Abdullah Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as Deviant Behavior: An Automated Black-box Noncompliance Checker for 4G LTE Cellular Devices. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*, Virtual, November 2021.

[30] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[31] Imtiaz Karim, Syed Rafiul Hussain, and Elisa Bertino. ProChecker: An Automated Security and Privacy Analysis Framework for 4G LTE Protocol Implementations. In *Proceedings of the 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021.

[32] Keysight. Anite Application Testing. https://www.keysight.com/us/en/products/wireless-network-emulators/4g-3g-2g-device-testing/anite-wireless-solutions/anite-application-testing.html.

[33] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. BaseSpec: Comparative analysis of baseband software and cellular specifications for L3 protocols. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual, February 2021.

[34] Hongil Kim, Dongkwan Kim, Minhee Kwon, Hyungseok Han, Yeongjin Jang, Dongsu Han, Taesoo Kim, and Yongdae 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, October 2015.

[35] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae 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.

[36] Sravan Kundojjala. Baseband Market Share Tracker Q2 2021: Qualcomm Leads with 52 Percent Revenue Share. https://www.strategyanalytics.com/access-services/components/handset-components/market-data/report-detail/baseband-market-share-tracker-q2-2021-qualcomm-leads-with-52-percent-revenue-share.

[37] Chi-Yu Li, Guan-Hua Tu, Chunyi Peng, Zengwen Yuan, Yuanjie Li, Songwu Lu, and Xinbing Wang. Insecurity of Voice Solution VoLTE in LTE Mobile Networks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, October 2015.

[38] Marc Lichtman, Roger Piqueras Jover, Mina Labib, Raghunandan Rao, Vuk Marojevic, and Jeffrey H Reed. LTE/LTE-A Jamming, Spoofing, and Sniffing: Threat Assessment and Mitigation. *IEEE Communications Magazine*, 54(4):54–61, 2016.

[39] Huang Lin. LTE REDIRECTION: Forcing Targeted LTE Cellphone into Unsafe Network. In *Hack In The Box Security Conference (HITBSec-Conf)*, 2016.

[40] H-A Loeliger. An introduction to factor graphs. *IEEE Signal Processing Magazine*, 21(1):28–41, 2004.

[41] Dominik Maier, Lukas Seidel, and Shinjo Park. BaseSAFE: Baseband SAnitized Fuzzing through Emulation. In *Proceedings of the 13th Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, Virtual, July 2020.

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

[43] Benoit Michau and Christophe Devine. How to not break lte crypto. In *ANSSI Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2016.

[44] Collin Mulliner, Nico Golde, and Jean-Pierre 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, August 2011.

[45] Collin Mulliner and Charlie Miller. Fuzzing the Phone in your Phone. In *Black Hat USA Briefings (Black Hat USA)*, Las Vegas, NV, July 2009.

[46] Navid Nikaein, Raymond Knopp, Florian Kaltenberger, Lionel Gauthier, Christian Bonnet, Dominique Nussbaum, and Riadh 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, September 2014.

[47] LLC Nuand. bladeRF. https://www.nuand.com/bladerf-2-0-micro/.

[48] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth downlink negative testing framework for LTE devices. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Boston, MA, August 2022.

[49] Shinjo Park, Altaf Shaik, Ravishankar Borgaonkar, Andrew Martin, and Jean-Pierre Seifert. White-Stingray: Evaluating IMSI Catchers Detection Applications. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, Canada, August 2017.

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

[51] Muhammad Taqi Raza, Fatima Muhammad Anwar, and Songwu Lu. Exposing LTE Security Weaknesses at Protocol Inter-Layer, and Inter-Radio Interactions. In *International Conference on Security and Privacy in Communication Systems*, pages 312–338. Springer, 2017.

[52] David Rupprecht, Kai Jansen, and Christina 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, August 2016.

[53] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking LTE on Layer Two. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[54] Altaf Shaik, Ravishankar Borgaonkar, N Asokan, Valtteri Niemi, and Jean-Pierre 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, February 2016.

[55] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre 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.

[56] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[57] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni 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.

[58] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis, 2014.

[59] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. Security Testing of GSM Implementations. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 179–195. Springer, 2014.

[60] Ralf-Philipp. 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, August 2012.

[61] Ralf-Philipp. Weinmann. Baseband exploitation in 2013: Hexagon challenges. In *PACSEC 2013*, Tokyo, Japan, 2013.

[62] Ben Wojtowicz. OpenLTE. *An open source 3GPP LTE implementation*, 2016.

[63] Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim. Hiding in Plain Signal: Physical Signal Overshadowing Attack on LTE. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[64] Jonathan S Yedidia, William Freeman, and Yair Weiss. Generalized belief propagation. *Advances in neural information processing systems*, 13, 2000.

[65] Chuan Yu and Shuhui Chen. On effects of mobility management signalling based dos attacks against lte terminals. In *2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC)*, 2019.

[66] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide 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, February 2014.

[67] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2021.

[68] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. In *Proceedings of the 2019 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Athens, Greece, October 2019.

Table A1: The list of firmware images used for evaluating BASECOMP.

Vendor	Arch	Nick	Model	Version	Date
Samsung	ARM	G950	Galaxy S8	G950NKOU5DUC1	Apr/2021
	ARM	G955	Galaxy S8+	G955NKOU5DUC1	Apr/2021
	ARM	G960	Galaxy S9	G960NKOU5FUJ1	Oct/2021
	ARM	G965	Galaxy S9+	G965NKOU5FUJ1	Oct/2021
	ARM	G970	Galaxy S10e	G970NKOS7FUJ1	Oct/2021
	ARM	G975	Galaxy S10+	G975NKOS7FUJ1	Oct/2021
	ARM	G977	Galaxy S10 5G	G977NKOS6FUJ2	Nov/2021
	ARM	G991	Galaxy S21 5G	G991NKOU3BUKF	Nov/2021
	ARM	G996	Galaxy S21+ 5G	G996NKOU3BUKF	Nov/2021
	ARM	G998	Galaxy S21 Ultra 5G	G998NKOU3BUKF	Nov/2021
MediaTek	ARM	Pro 7	MEIZU Pro 7	MOLY.LR11.W1630.MD.MP.V17.P48	Sep/2017
	MIPS	A31	Galaxy A31	MOLY.LR12A.R3.TC10.6M.PR.KR.SP.V2.P29	Dec/2021
	MIPS	A31 (Latest)	Galaxy A31	MOLY.LR12A.R3.TC10.6M.PR.KR.SP.V3.P32	Feb/2023
	MIPS	A03s	Galaxy A03s	MOLY.LR12A.R3.TC10.6M.A03S.NA.PR.SP.V3.T1212	Feb/2023
	MIPS	A145	Galaxy A145	MOLY.LR12A.R3.TC10.6M.A14.PR.SP.V1.P5	Feb/2023
srsRAN	x86	srsran	-	22.04.1	Aug/2022

A Acronyms

3GPP	Third Generation Partnership Project
AKA	Authentication and Key Agreement
AP	Application Processor
BP	Baseband Processor
EMM	EPS Mobility Management
GUTI	Globally Unique Temporary Identity
IE	Information Element
IMEI	International Mobile Equipment Identity
IMEISV	IMEI with Software Version
IMSI	International Mobile Subscriber Identity
MAC	Message Authentication Code
MitM	Man-in-the-Middle
MME	Mobility Management Entity
NAS	Non Access Stratum
RRC	Radio Resource Control
SIM	Subscriber Identity Module

B Dataset

Table A1 shows a list of firmware binaries used for evaluation. We describe each column as follows. The "Vendor", "Arch" and "Model" columns are literally the vendor, arch and model of the firmware. To label the firmwares in an easier way, we use the names in the column "Nick". The "Version" and "Date" columns indicate the name and the release date of the version we used. Lastly for srsRAN, as it is an open-source project, we only use it release date of the version used is denoted.

Table A2 shows the lines of code for vendor- and model-specific files. Due to the space limit, we omitted its detail, but BASECOMP also has a yaml configuration file for vendors. This file contains several pieces of information for analysis, including the location of Python-based analysis module and the firmware's architecture.

Table A2: Lines of code for vendor- and model-specific files.

Vendor	Arch	Model	LoC (Vendor)	LoC (Model)
Samsung	ARM	G950	14 (Python) 8 (yaml)	10
		G955		10
		G960		11
		G965		11
		G970		11
		G975		11
		G977		11
		G991		11
		G996		11
		G998		11
MediaTek	ARM	P25	18 (Python) 7 (yaml)	12
	MIPS	A31	26 (Python) 7 (yaml)	13
		A31 (Latest)		13
		A03s		12
		A145		13
srsRAN	ARM	-	15 (Python) 7 (yaml)	10