A Static Workflow for Cryptography Verification in 4G/5G Basebands

Lorenzo Lucca[1], Sara Da Canal[2,3], Giuseppe Bianchi[2,3] and Andrea Visconti[1,*]

[1]University of Milan
[2]CNIT National Network Assurance and Monitoring (NAM) Lab, Rome, IT
[3]University of Rome "Tor Vergata", Rome, IT

Abstract

In this paper we detail a workflow devised to identify and verify cryptographic implementations within baseband firmware. The approach combines constant-guided function discovery, semantic comparison against 3GPP specifications, parameter bound extraction, and library fingerprinting. We apply the method to three commercial firmware images from Samsung (Shannon) and MediaTek devices, finding and validating vendor implementations of KASUMI, SNOW-3G, AES, and 4G/5G integrity and confidentiality algorithms. Our analysis reveals security-relevant issues, including incorrect S-Box indexing in KASUMI and a firmware update failure caused by a malformed signature field. We further extract a 1792-bit RSA limit in Shannon and identify an outdated OpenSSL 1.0.2c bundle in MediaTek, implying exposure to unpatched CVEs. Results demonstrate that structured cryptographic verification in baseband firmware is not only feasible but essential for supply-chain transparency and long-term security audits.

Keywords

Baseband firmware, Cryptographic verification, Static analysis, 4G/5G security, Reverse engineering

1. Introduction

Modern societies depend on secure cellular infrastructures and connectivity for personal communication, economic services, critical logistics, and essential public functions. At the core of every cellular device lies the baseband processor, the component responsible for implementing the radio access protocol stacks and guaranteeing end-to-end connectivity.

Far from being a simple modem, the baseband is an isolated computing subsystem, running its own real time operating system, and executing hundreds of protocol-specific tasks. Its firmware is built upon 3GPP specifications [1], which span hundreds of documents and thousands of pages written in natural language. Such specifications are inherently complex, occasionally ambiguous, and therefore prone to misinterpretation, making correct and consistent implementation particularly challenging [2, 3]. Firmware images routinely exceed hundreds of megabytes, integrating protocol handlers for all network generations from 2G to 5G within a single binary. The resulting code base is not only large, but also historically layered: due to the need for backward compatibility, legacy protocol functions must coexist with modern 4G/5G security mechanisms, so that code relative to older communication network generations is kept inside new firmware, adding complexity and increasing the possibilities of bugs.

Experience in telecom security repeatedly shows that highly stratified codebases tend to accumulate technical debt over time. Past analyses in neighboring subsystems, such as SIM toolkits, AT command handlers, and radio interface layers, have exposed outdated cryptographic libraries, unused legacy code, and vendor patches applied unevenly across generations [4, 5]. Unlike other critical software domains, baseband firmware is typically distributed without a Software Bill of Materials [6] or dependency

Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), February 09-13, 2026, Cagliari, IT
*Corresponding author.
lorenzo.lucca@studenti.unimi.it (L. Lucca); sara.dacanal@cnit.it (S. D. Canal); giuseppe.bianchi@uniroma2.it (G. Bianchi); andrea.visconti@unimi.it (A. Visconti)
0009-0003-2263-2420 (L. Lucca); 0009-0006-8928-7096 (S. D. Canal); 0000-0001-7277-7423 (G. Bianchi); 0000-0001-5689-8575 (A. Visconti)

manifest, preventing operators and auditors from knowing which cryptographic libraries or third-party modules are currently embedded, which versions are deployed, and whether maintained variants exist.

This raises a foundational question: *how can a security analyst assess the cryptographic soundness of baseband internals when library composition, key parameters, and implementation details are opaque by design*?

Direct reverse engineering of full baseband images is rarely feasible in practice. Huge sizes, architectural diversity, stripped symbols, proprietary loaders, multi-generation protocol layering, and vendor-specific crypto implementations make holistic assessment prohibitively expensive. Even when partial reversing succeeds, deriving assurance on algorithm correctness, parameter boundaries, and update-signature logic remains burdensome without systematic methodology.

In this paper, we address this challenge by presenting a structured workflow for identifying, analyzing, and verifying cryptographic implementations inside baseband firmware. Our approach combines constant-guided function discovery, semantic comparison against 3GPP specifications, parameter bound extraction, and library fingerprinting, enabling targeted validation without full firmware decompilation.

Applying the workflow to Samsung Shannon and two MediaTek platforms, we (i) verify vendor implementations of KASUMI, SNOW-3G, AES, and 4G/5G integrity and confidentiality functions; (ii) extract cryptographic parameter constraints, including a 1792-bit RSA limit; (iii) identify an operational defect in firmware-update signature recognition; and (iv) fingerprint an outdated OpenSSL 1.0.2c deployment, highlighting long-term patching gaps. Our results show that structured static assessment of baseband cryptography is feasible and constitutes a necessary step toward verifiable security in cellular devices.

The remainder of this paper is organized as follows. Section 2 provides background and reviews related work. Section 3 describes the workflow and methodology. Section 4 details the system setup, while Section 5 presents the analysis of cryptographic functions and libraries. Section 6 reports the testing activities and results. Finally, Section 7 concludes the paper and outlines future work.

2. Background and Related Work

2.1. Baseband Processors

A baseband processor (BP) is a dedicated chip in a wireless device that handles all radio communication functions, converting digital data into radio waves for transmission and vice versa, and managing all layers of the related network protocols. Baseband processors usually run a real-time operating system (RTOS) as their firmware. In this kind of system, the majority of functionality is implemented as a large number of tasks that implement either OS functionality or cellular-specific functions and communicate using message queues. Usually, every layer of a specific network protocol is managed by a different task. As illustrated in Figure 1, a baseband processor communicates both with the application processor — i.e., the main chip of the phone or other device on which the BP is used — using AT commands [7], and with a series of peripherals [8].

The main functionalities of a Baseband are related to communication according to every network generation, from 2G to 5G, and it manages all their protocols and all security aspects. For cellular systems, this means treating the SIM as a peripheral that is contacted whenever keys are necessary [9]. On the other hand, a Baseband is also responsible for auxiliary functionalities, like managing over-the-air updates or GPS connectivity.

Baseband firmware is closed source, and analyzing chips from different vendors requires several adaptations, considering that each one is based on a different ISA. We focus on Mediatek and Samsung firmware (Shannon), which together covered almost half of the BP market in 2024 and 2025 [10], and which have been the focus of several researches. Another major player is Qualcomm, which has been excluded from security researches so far because their firmware is based on Hexagon, a proprietary architecture, and is compressed [11].

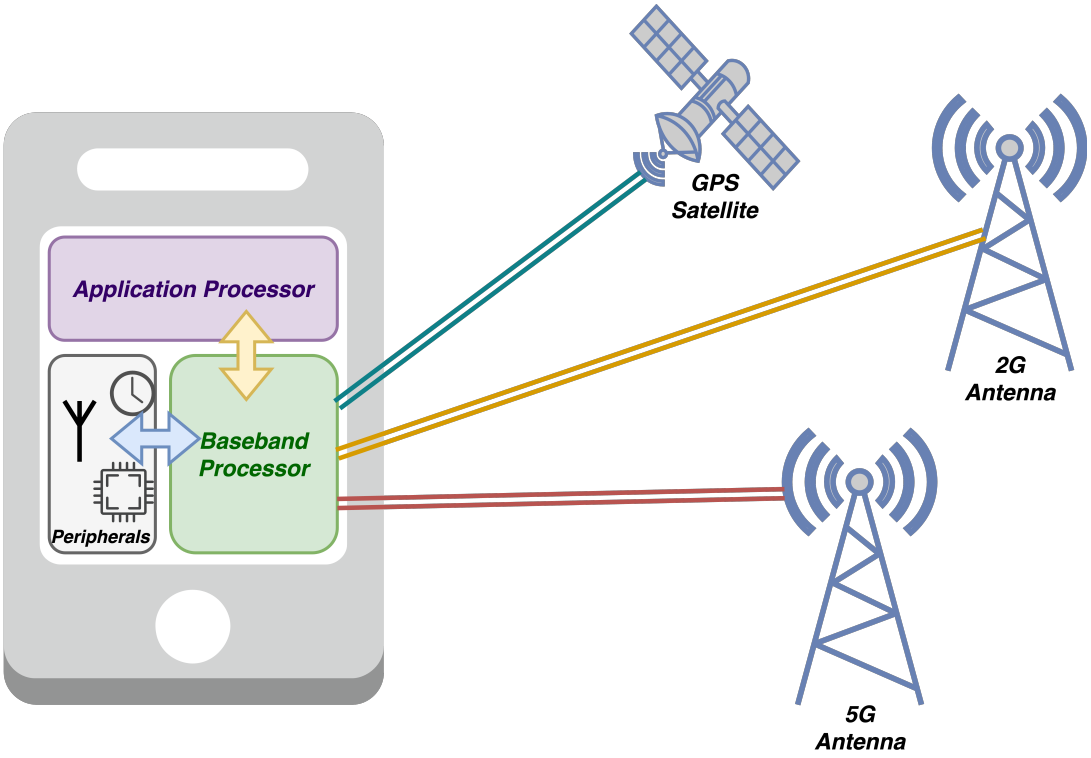

Figure 1: Smartphone general architecture

2.2. Related Work

Different works related to baseband security focus on protocol fuzzing over the air, adopting a black-box approach and avoiding firmware reversing, like [12], [13], and [14]. Projects choosing to test over the air highlight the numerous difficulties in analyzing a complex baseband firmware, but also the limitation of the over-the-air approach that can be overcome by static and dynamic analysis.

On the other hand, there have been different efforts whose focus was on detecting bugs or inconsistencies by reversing, usually by choosing an interesting point based on debug strings and manually reversing from there, thus lacking a systematic and automated approach [15], [16].

There are some works whose focus is on choosing a particular aspect of a baseband firmware and developing static analysis tools to aid in analyzing some specific feature, like BaseMirror [17], which automates AT command discovery, or BaseComp [18], which narrows the functions responsible for 4G and 5G integrity protection. The latter has demonstrated the presence of flawed implementations of integrity functions, which are critical for device security, through semi-automatic methods for identification based on factor graphs constructed from specification. Our work adopts a similar approach, focusing on identifying generic cryptographic functions instead of specific ones. Another major focus in baseband firmware static analysis is systems that facilitate comparing functions between firmware of different chips or different versions, allowing manual analysis to be performed only once, like BaseSpec [19].

Another thread of baseband analysis is emulation, with Firmwire [8] as state of the art. Emulation still relies on static analysis, because to obtain a functional emulator heavy reversing is necessary.

In the context of general cryptographic analysis of binaries, there exist works such as CryFind[20] that employ static analysis to automate the identification of cryptographic functions within binaries. These works search for known strings and constants, and verify their presence with different encodings to improve the match rate. Alternative methods are predicated on pattern matching through rules, including those based on YARA[21]. The identification of cryptographic algorithms through their structural characteristics has been a subject of research, with proposals relying on the sequence of instructions to be executed, as evidenced by Grap[22].

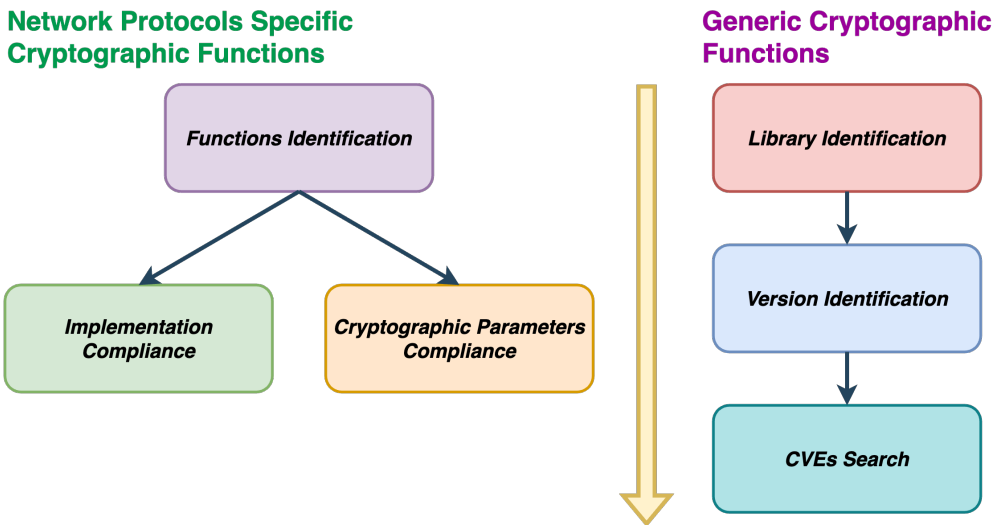

Figure 2: Cryptographic Security Verification Main Workflow

A recent development in the field of automating the identification and analysis of cryptographic function implementations involves the use of LLMs. In this context, both methods based on generic LLMs, such as ChatGPT [23], and methods based on purpose-built LLMs, as in the case of FoC [24], have been employed. These methods take advantage of the generalization capability of LLMs to reduce the manual effort required for analysis, thereby facilitating the process.

3. Workflow and Methodology

A comprehensive analysis of cryptographic functions' soundness and security needs to consider different aspects. Cellular protocols' integrity and encryption capabilities rely on specific cryptographic functions, exclusively employed for cellular communications. No state-of-the-art library exists, and vendor-specific implementations are common. However, basebands may require cryptographic functions for additional purposes, such as verifying signatures of critical data during firmware updates. Implementing generic cryptographic functions from scratch would be impractical and time-consuming, given the availability of widely used and tested libraries.

As depicted in Figure 2, due to the different starting knowledge, the process for analyzing vendor-specific or library functions is different: when verifying the correctness of the former, function identification is a preliminary step before manually comparing the function implementation against the specification and verifying parameter compliance with security standards, while for the latter, focusing on the used library can lead to better results. For this purpose, we leveraged a similarity-based method, checking the binary for a set of well-known cryptographic libraries and manually comparing found implementations to the source code in order to pinpoint a specific version. Once the library and version are known, searching for security-relevant issues of even unpatched vulnerabilities is a straightforward process.

For analyzing vendor-specific functions, some automation is necessary due to the huge size of the analyzed binary. We employed a method based on cryptographic constants, expected call graphs, and strings to identify promising functions to analyze; afterward, we select a number of significant checks to perform on such functions:

- Semantic compliance with the specification
- Parameter length and boundary checks
- Constants and keys length and security

By defining a set of verifications to perform, static analysis can be sped up, and a comprehensive security audit of cryptographic functions can be performed.

4. System setup

To enable the analysis of cryptographic functions implemented in firmware, a number of preliminary operations were necessary. First, a firmware image must be acquired whose functions will be tested for correctness. Then, preliminary automated analysis and decompilation can be conducted to create a more comprehensible representation of the examined functions and extract useful data for subsequent analysis stages.

4.1. Firmware Acquisition

Baseband firmware is typically packaged inside the chip, so it must be obtained before analysis can begin. One possible approach is to leverage online update packages, a strategy that has been documented for both Shannon [25] and MediaTek [26] processors. These techniques usually involve extracting a series of nested archives and decompressing files. Another method involves extracting the firmware directly from a physical device using adb [27] to read and write to the file system. However, this method requires root privileges on the phone. Samsung phones also offer the option to extract RAM dumps — i.e., snapshots of the device memory containing both booted firmware and data.

4.2. Firmware Loading and Preliminary Analysis

In order to analyze the functions contained in the firmware image, the image must first be correctly loaded into a reverse engineering framework such as IDA [28] or GHIDRA[29] and a preliminary automatic analysis must be performed. The first step in loading the binary into the software is specifying the parameters related to the target processor architecture of the firmware. The chosen program must naturally support the firmware architecture, either through official support or community-developed extensions. To correctly load the binary, appropriate loaders are also necessary. Loaders interpret the firmware's specific format and map it to the software's internal representation. Such tools exist for Mediatek firmware [26] and Shannon firmware[25].

Once the firmware is loaded into the software, a preliminary automatic binary analysis phase must be performed. During this phase, some of the firmware's functions are identified. If the configuration allows, strings and data contained in the binary can also be detected. Then, it is possible to decompile the identified functions. Additional functions originating from calls made by the decompiled code are discovered during this process, as well as other data used by these functions.

5. Static Analysis

In this section, we focus on identifying the cryptographic functions implemented in the firmware, verifying that the cryptographic functions specific to the baseband domain — e.g., Snow-3G, KASUMI, F8, F9 — were implemented according to specifications, ensuring that the cryptographic parameters used followed international security standards, and analyzing the libraries employed to determine their versions and thereby gain access to known vulnerabilities documented in the literature for such versions. Various methods were used throughout these phases to establish a functional and robust workflow for analyzing cryptographic functions in firmware.

5.1. Crypto Functions Identification

One of the main challenges in binary reverse engineering is function identification. In order to analyze a given function, its implementation must first be located within the binary. When debug symbols are available, the starting address can be retrieved by searching through function signatures. Function signatures typically comprise the protocol name or module identifier and the function name. In such cases, function identification is straightforward. However, when symbols are unavailable, alternative methods must be used to locate functions.

Due to the large number of functions implemented within the baseband firmware, an exhaustive search is infeasible. We have used two methods to identify and locate cryptographic functions of interest: one based on string searches and the other based on the constants used by the functions. These methods are used to identify functions that are specifically implemented by vendors and do not belong to a library. Methods for analyzing library functions are described in a separate chapter. These methods have already been widely adopted by previous studies, and have proven effective in identifying functions [20].

In order to identify a function based on strings, it is first necessary to identify all of the strings contained within the binary. Then, it's possible to search for strings that can be linked to functions related to the target function. High-level functions, which may use cryptographic functions, usually have a logging mechanism to aid in debugging and status checking. These logging functions use formatted strings with a similar structure that typically contains the protocol name or an identifier at the beginning, followed by the function's name. Searching for the function or protocol name in the strings allows you to locate every logging entry in which the target function or the protocol to which the function belongs appears. Then, it's possible to analyze the function in which the identified string is used to further identify the target function. In most cases, this process can be automated [30], generating names for most high-level functions in the firmware. Otherwise, manual analysis can be employed, which involves examining string constants within each function to identify references to function names or protocol-related information, such as source paths.

Another widely adopted method is to identify functions based on the constants used. Cryptographic functions often use constant values and tables, such as S-boxes. One way to identify functions that use these constants is to search the binary for the constant's signature. Once identified, reverse engineering frameworks typically allow examination of data address references within functions. After generating a list of functions that use the constant, a straightforward analysis can be performed to verify the function's semantics and compare it to the target function.

We have used a method to partially automate the identification of cryptographic functions. First, a directed graph representing the target function structure is built using the function structure (an internal call graph) and the constants used by each subroutine. The algorithm then searches for the set of constants used by the subroutines. If these are found inside the binary, the algorithm proceeds to build the reference graph of the constant set until the target function height is reached. Then, the target function structure and the generated structure are compared for matches using subgraph isomorphism. The main flaw of this approach is that it only uses the function's structure and the constants used to characterize it, without checking the semantics. This renders the approach very fast, but it also produces many false negatives in cases involving functions with single constants and simple function structures. For example, this occurs when identifying AES.

5.2. Analysis of Crypto Functions

We used static analysis to determine the correctness of the cryptographic function implementations. The workflow used for each function was as follows:

- identify the target function;
- obtain the function specification from the organization;
- compare the implementation with the specification;
- identify possible security issues regarding the implementation.

The primary objective of this phase has been to compare the implementations of functions against their respective specifications. Consequently, the decompiled code produced by the selected software's decompiler undergoes analysis to determine the function's behavior. In the event that the observed behavior does not align with the established specification, the function is considered to be problematic. More precisely, the initial phase of the process entails the mapping of code segments that have been decompiled to the discrete operations constituting a given function, in accordance with the established

specifications. In this manner, after isolating the operations, it is possible to verify that the two semantics—theoretical and implemented—correspond. It is imperative to methodically examine the potential outcomes of specific cases and the various possible input dimensions. Cryptographic functions are defined from a mathematical perspective, whereas implementations are contingent upon the utilization of the underlying hardware. For instance, if the specification of a function stipulates that it will accept a 16-bit input, then it is highly probable that the implementation on a 32-bit architecture will employ a 32-bit value to represent it. While this does not inherently constitute a problem, it is imperative to ensure that only values the function can process are passed to it. To clarify, if a function employs an S-Box that maps a 16-bit value to a 16-bit value, and this S-Box is implemented as a lookup table, passing a value represented as 32 bits to this S-Box could result in memory access outside the S-Box boundaries, potentially reading arbitrary data.

Previous studies [18] have documented examples of functions that behave unexpectedly when given certain inputs, leading to security issues that expose the device to attacks. Even if a function matches its specification, further analysis is needed to determine if security issues could arise from its implementation. Examples of this include not checking pointers for null values, using unallocated memory, and failing to manage errors correctly.

It is then necessary to check the functions that call the target function. An incorrect calling procedure, failure to set up parameters correctly, and other implementation errors can result in a function that cannot behave properly. Thus, it is necessary to analyze every function that interacts with the target function and study how they interact to verify that the expected behavior is always guaranteed. Static analysis is slow due to its manual nature, and it doesn't scale. Therefore, a selection of the functions to analyze is needed, even though this implies a level of uncertainty regarding the correctness of the implementation. Analyzing the entire firmware using this approach would be extremely expensive in terms of time and resources.

5.3. Analysis of Crypto Parameters

A problem arises when a cryptographic function is implemented correctly but used with insecure parameters. For example, using an RSA function with a key of insufficient length results in an insecure system, as it can be easily attacked to compromise the integrity and confidentiality. A crucial part of the analysis is checking these parameters to obtain more information. Usually, analyzing cryptographic parameters means analyzing the key length if the algorithm uses a variable-length key. This is often done by providing an upper bound on the possible key length. The process of extracting information about cryptographic parameters is as follows:

- Analyze the target function for potential parameter limits;
- Analyze the calling functions for potential parameter limits during setup;
- Analyze the stored parameters in the binary.

The analysis of the target function for parameter limits involves checking the decompiled code to verify the presence of structural limits for parameter size. For example, a hard-coded value may be used to represent the size of a parameter, remaining fixed regardless of the parameter's actual size. Such hard-coded values were identified in some of the analyzed functions, where they are used to bound loops during parameter copying. These explicit parameter limits are generally the easiest to detect, and their security implications can typically be assessed with minimal effort.

Another potential constraint on parameter sizing may arise from calling functions that pass working parameters to cryptographic routines under analysis. Since these callers allocate, manage, and pass parameters, they are responsible for the life cycle of the data structures containing them. Therefore, by examining the calling functions, it is possible to extract useful information from memory allocation and deallocation operations, as well as from more general structure-handling operations, such as assignments to structure fields. In particular, memory allocation and deallocation can provide an upper bound on the size of the structure containing the parameter and, consequently, an upper bound on the parameter itself. Furthermore, structure management operations may reveal details about the

structure's layout. For instance, printing routines or assignments to specific fields may reveal the structure's individual fields and their corresponding offsets. In most cases, parameters managed and passed by calling functions are retrieved from device memory, such as flash memory, for example cryptographic keys are stored on SIM cards. In other cases, some of these parameters may be stored directly in the data section of the firmware binary instead. In these situations, it is possible to identify the memory region where the parameters reside and analyze their structure. Examining this structure allows for an exact determination of the size of the contained parameters and enables the execution of security checks to detect potential issues related to the specific parameters in use.

Once bounds have been determined for the parameters used by the cryptographic function, it is possible to compare them with the recommendations provided by current standards. This comparison verifies whether the employed parameters conform to the prescribed requirements. In the event that the parameters in use have been extracted directly, it is imperative to analyze them to ensure that they do not exhibit security issues, such as known factorizations in the case of RSA keys.

5.4. Libraries Identification

The cryptographic functions that were manually analyzed in this work were vendor-specific implementations for baseband use cases. These functions were exclusively employed for cellular communications. However, basebands may require cryptographic functions for additional purposes, such as verifying signatures of critical data during firmware updates. The implementation of generic cryptographic functions from scratch would be impractical and time-consuming, given the existence of libraries that already provide the necessary functionalities. Given the focus of the analysis of firmwares in this work on cryptographic functions, particular attention was devoted to libraries implementing cryptographic capabilities.

A manual analysis of all the functions implemented by these libraries would be both slow and inefficient. The majority of these functions are not used by the firmware, and the remainder typically depend on runtime configuration. Therefore, they cannot be properly analyzed through static analysis alone. To illustrate, within the OpenSSL framework, the majority of cryptographic operations are encapsulated into discrete modules, designated as "engines." These modules facilitate the utilization of diverse implementations for a given operation. In the context of static code analysis, the identification of the implemented function can pose significant challenges. This is due to the possibility that the function identifier may not be stored in the binary, resulting in its inability to be determined. Furthermore, this process can be rendered excessively time-consuming, hindering its practical application. The widespread adoption of these libraries has the potential to accelerate the analysis process. In the case of open-source libraries, the source code and documentation can be examined to determine the security of the implemented functions and their conformance to specifications. In such cases, the analysis of cryptographic functions becomes more efficient, although the source code of these libraries is not always straightforward to interpret due to preprocessor directives that may alter the generated code based on configuration flags.

Another advantage of the pervasive utilization of these libraries across a myriad of software applications is the accessibility of security issue reports from other users and organizations. Leveraging these reports enables a more efficient analysis approach: verifying the library version implemented in the firmware and then querying public vulnerability databases to identify potential issues that may affect firmware operation.

The initial step in this process is to identify whether the target library is contained in the firmware. A number of function identification methods were employed, including:

- Debug symbol-based methods;
- Hash-based methods — e.g., IDA F.L.I.R.T.[31] and Ghidra Function ID;
- Function similarity-based methods — e.g., Ghidra BSim[32].

Each method has specific strengths and weaknesses. Symbol-based approaches depend on the availability of debug symbols in the distributed binary, which cannot be assumed. Furthermore, they require either

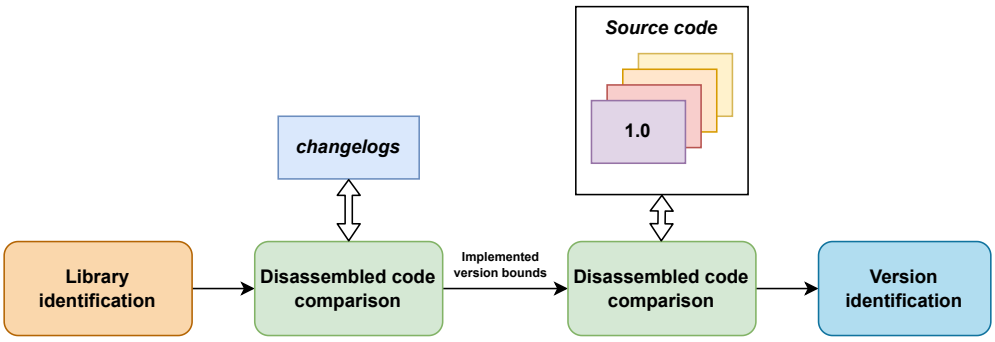

Figure 3: Manual identification of library version

be familiar with the library's naming scheme or have to access to a comprehensive function name list. Although this approach can confirm the presence of a library, it cannot directly identify the version. Thus, a subsequent code analysis remains necessary to verify the specific version implemented.

Hash-based methods, such as IDA F.L.I.R.T. and Ghidra FunctionID, operate by maintaining a comprehensive database of function signatures derived from hashes of the first bytes of functions. When analyzing a binary, the hashes in the database are compared against the computed hash of each function; an exact match indicates successful function identification. The primary limitation of this approach comes from the requirement that library functions hashes be computed from a binary compiled with the same configuration used for the firmware. Differences in target architecture, compiler choice, or compiler configuration may result in hash mismatches even when the library is actually present. Consequently, this approach proves ineffective without knowledge of these compilation parameters. However, when such details are available, comparing hashes computed from a library binary generated with the specified parameters against firmware function hashes enables precise version identification. The fundamental challenge of this method lies in its lack of flexibility: achieving reliable function matches would require computing hashes for all versions of the target libraries across all possible compilation configurations, including different compilers.

An alternative approach employed for identifying library functions was the use of similarity-based methods. The underlying principle of this approach is to use the decompiler to generate a feature vector for each function contained in the binary. These features represent the control and data flow of the function. The feature vector is then normalized to isolate the function's behavior from its specific implementation. Feature vectors are compared using cosine similarity metric, which enables functions with similar functionality but different implementations to achieve high similarity scores. This approach allows comparison of functions found in the binary with library functions regardless of differences in compilers or even target architectures.

Once the library has been identified, its version must be verified. The process of identifying the version can be executed through two distinct methodologies, depending on the available information regarding the firmware compilation process. If the compiler used is known, different versions of the library can be compiled and binary comparisons can be performed to identify differences using plugin tools available within the reverse engineering framework. This approach enables version ranking even when compilation option varies, ordering candidates from most probable (showing the greatest similarity) to least probable (showing substantial differences, which are attributable not only to compilation processes but also to actual functional changes and different available features).

If the compiler used to generate the firmware binary cannot be determined, the version can be verified by manually comparing the decompiled functions with the source code. Specifically, one can compare the functions present in the binary with those in the reference version of the library. For instance, one can check for the addition of new functions that implement new behaviors. The objective of this initial phase is to identify two reference versions: a lower bound version that lacks features implemented in the firmware functions, and an upper bound version that contains features not supported by the library

Table 1
Summary of our testing activity

Firmware Sample	Activities	Security Concerns
Shannon	Analysis of cryptographic function implementations (KASUMI, SNOW-3G, AES, RSA) and identification of cryptographic parameter sizes	RSA maximum key length below recommended security standards
Mediatek 1	Analysis of cryptographic function implementations (KASUMI, SNOW-3G, AES, RSA) and identification of cryptographic parameter sizes	Potential incorrect memory access in the KASUMI implementation
Mediatek 2	Analysis of cryptographic function implementations (KASUMI, SNOW-3G, AES, RSA, DSA, ECDSA), identification of cryptographic parameter sizes, and OpenSSL version detection	Potential update failure due to a typo in a field name; OpenSSL version 1.0.2c is no longer supported

version used in the firmware. During this phase, it is not necessary to analyze the library's source code (see Figure 3). Examining the version changelogs and comparing it to the functions implemented in the firmware is sufficient to identify version bounds and thus speed up the process. After identifying a subset of candidate versions, the actual version used must be isolated. This stage requires analyzing the public source code of the candidate versions and comparing them with the decompiled code of the functions contained in the binary. Here, version control systems such as Git are crucial, as they enable source code comparison between versions, allowing to focus solely on functions that change across specified versions. These techniques allow for the rapid identification of the library version even when automated analysis is not feasible. Once the library version has been identified, the next step is to verify known vulnerabilities. This can be done by querying vulnerability databases with the library name and version number. Examples of these databases include the National Vulnerability Database (NVD) and the Snyk Vulnerability Database. However, other databases may also be consulted in the same way.

6. Testing activity

The methods described in this paper were tested (see Table 1) on three different firmware images: one from the Shannon family, developed by Samsung, and two developed by MediaTek. The process for each firmware image involved first identifying the cryptographic primitives implemented for baseband functions, primarily KASUMI, SNOW-3G, and AES. Then, the focus shifted to the algorithms utilizing these primitives, including F8, F9, EIA1, EEA1, EIA2, and EEA2.

Shannon Firmware: The analyzed cryptographic functions (KASUMI, SNOW-3G, AES, and RSA) were found to be fully compliant with their specifications. An upper bound was determined for the maximum key size usable with RSA. An in-depth analysis of the functions managing the key structure revealed a function that compute the ASN.1 format representation of the private key. This function uses 7 buffers of 8 words each, each having 32 bits, to store the key components obtained from the functions. Using this information, we are able to compute the maximum key size: 1792 bits. This result was then cross-referenced with other functions using the key to verify this maximum size. The recommended RSA key length is 2048 bits [33], consequently, the key used in this specific RSA implementation does not satisfy this requirement.

MediaTek firmware (first): This firmware was selected and used in the Motorola Moto Edge 2022 device. It is a MediaTek firmware for a MediaTek Dimensity 1050 SoC. We analyzed the same cryptographic functions as in the Shannon firmware. These functions presented no issues apart an implementation of KASUMI. In such implementation we identify a potential vulnerability in the FI function when 32-bit values are passed and an incorrect memory access occurs, creating a potential security risk. The other functions were found to be compliant with specifications during comparative analysis, despite having

a different implementation than what was found in Shannon. No bounds on specific cryptographic parameters were found.

MediaTek firmware (second): For this firmware, the analysis goal was expanded to examine not only baseband-specific cryptographic functions, but also the libraries using the described methods. The specific cryptographic functions were found to comply with the specifications, though the error handling was inadequate or entirely absent. Implementations of DSA, ECDSA, and RSA through OpenSSL were verified as well. No issues were detected for DSA or ECDSA, as only signature verification methods were used. RSA is implemented through OpenSSL as well, and no limit on key size was detected. Through analysis of the public key management functions, 12 public keys used as trusted developer keys were extracted from the firmware binary. Security checks were performed on these keys. No other implementation errors were found except in a data signature verification function for critical data, used during firmware updates, where a typo in the field "crticalDataSign:" prevents correct recognition of the field "criticalDataSign:". This may prevent firmware updates if the received structure does not replicate the existing error.

The OpenSSL version implemented in the firmware is 1.0.2c. This version was found using the methods described above. It is important to identify the precise patch level to facilitate comprehension of the vulnerabilities that have been addressed and the issues that remain unresolved. This particular version has been out of support since January 1, 2020, and consequently no longer receives updates. It should be noted that not all published vulnerabilities pertaining to this version pose security concerns, as the firmware implementation utilizes only a limited subset of the library's functions. However, the deployment of these libraries is no longer supported, indicating a lack of regular updates to these software components.

The adoption of three different firmware configurations made it possible to verify the versatility and adaptability of our methodology. The main difference in the analysis process appears at the beginning, during the environment setup phase, since different firmware requires different initial steps. Function identification was carried out both by exploiting constants and function structures and by searching for names in debug symbols when functions were harder to identify. Using these approaches, all predefined target functions were successfully identified in every firmware. Once identified, the code analysis of the functions and the study of cryptographic parameters were performed in the same way across all firmware.

The methods that successfully identified libraries in the firmware were the search for library functions in debug symbols and function similarity analysis using BSim. In contrast, comparing function hashes with databases of known signatures did not produce the interesting results due to differences in library versions and compilers. Notice that all these methods were tested on every firmware. On the contrary, due to time constraints, library version identification was carried out on only one firmware. Although manual, the method proved to be effective and appears applicable to other firmware without issues.

7. Concluding Remarks

In this paper, we presented a methodology aimed at evaluating the correctness of the implemented cryptographic functions. The proposed approach provides a structured framework for analyzing firmware at the baseband level, enabling a detailed assessment of cryptographic implementations and their compliance with expected behaviors.

A key strength of the methodology lies in its independence from the specific baseband firmware family under analysis. Although the approach was validated on three different firmware configurations in this study, the results demonstrate that it is generally applicable and can be extended to other baseband firmware with minimal adaptations.

Furthermore, the adopted approach enables the automated identification of cryptographic functions, significantly reducing the amount of manual effort required during analysis. This level of automation not only improves scalability but also facilitates the detection of potential weaknesses or vulnerabilities within cryptographic implementations.

As part of our testing activities, we applied the proposed method to three commercial baseband firmware images, one Samsung (Shannon) and two MediaTek. The analysis allowed us to identify and validate vendor implementations of major cryptographic algorithms and to uncover some security issues, including implementation errors and the use of outdated cryptographic components. These results confirm the effectiveness of our approach for systematically testing baseband firmware and highlight the importance of structured cryptographic verification for long-term security assessments.

Acknowledgments

This work was partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP F83C22001690001 and E83C22004640001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART") and project SERICS (PE00000014) under the NRRP MUR program funded by the EU—NextGenerationEU. It was also partially supported by the Digital Europe Programme EDIH I-NEST "Italian National hub Enabling and Enhancing networked applications & Services for digitally Transforming SMEs and Public Administrations" G.A. 101083398.s. Andrea Visconti is member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Declaration on Generative AI

During the preparation of this work, the authors used GPT-3.5 in order to: grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] 3GPP, 3rd generation partnership project (3gpp), https://www.3gpp.org, 2026.

[2] Y. Chen, Y. Yao, X. Wang, D. Xu, C. Yue, X. Liu, K. Chen, H. Tang, B. Liu, Bookworm game: Automatic discovery of lte vulnerabilities through documentation analysis, in: 2021 IEEE Symposium on Security and Privacy (SP), IEEE, 2021, pp. 1197–1214.

[3] I. Karim, S. R. Hussain, E. Bertino, Prochecker: An automated security and privacy analysis framework for 4g lte protocol implementations, in: 2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS), IEEE, 2021, pp. 773–785.

[4] J. K. Manda, Cybersecurity strategies for legacy telecom systems: Developing tailored cybersecurity strategies to secure aging telecom infrastructures against modern cyber threats, leveraging your experience with legacy systems and cybersecurity practices, Leveraging your Experience with Legacy Systems and Cybersecurity Practices (January 01, 2017) (2017).

[5] K. Mayes, T. Evans, Smart cards and security for mobile communications, in: Smart Cards, Tokens, Security and Applications, Springer, 2017, pp. 93–128.

[6] L. Bendix, A. Göransson, A comprehensive view of Software Bill of Materials, Technical Report, Technical report, Dept. Comput. Sci., Lund Univ., Sweden, 2023.

[7] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, C. Raules, P. Traynor, H. Vijayakumar, L. Harrison, A. Rahmati, M. Grace, K. R. B. Butler, ATtention spanned: Comprehensive vulnerability analysis of AT commands within the android ecosystem, in: 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, Baltimore, MD, 2018, pp. 273–290. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/tian.

[8] G. Hernandez, M. Muench, D. Maier, A. Milburn, S. Park, T. Scharnowski, T. Tucker, P. Traynor, K. R. B. Butler, FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware, in: Symposium on Network and Distributed System Security (NDSS) , 2022.

[9] T. P. Lisowski, M. Chlosta, J. Wang, M. Muench, SIMurai: Slicing through the complexity of SIM card security research, in: 33rd USENIX Security Symposium (USENIX Security 24), USENIX

Association, Philadelphia, PA, 2024, pp. 4481–4498. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/lisowski.

[10] C. Research, Smartphone application processor (ap) / system-on-chip (soc) vendor shipment share worldwide in 2024 and 2025, https://counterpointresearch.com/en/insights/global-smartphone-apsoc-market-share-quarterly, 2025.

[11] L. Glockow, R. Shriwas, B. Produit, Securing the airwaves: Emulation, fuzzing, and reverse engineering of iphone baseband firmware, in: TROOPERS25 Conference, 2025. Talk presented at TROOPERS25.

[12] M. E. Garbelini, Z. Shang, S. Luo, S. Chattopadhyay, S. Sun, E. Kurniawan, 5ghoul: Unleashing chaos on 5g edge devices via stateful multi-layer fuzzing, IEEE Transactions on Dependable and Secure Computing 22 (2025) 6230–6247. doi:10.1109/TDSC.2025.3582093.

[13] J. Yang, Y. Wang, T. X. Tran, Y. Pan, 5g rrc protocol and stack vulnerabilities detection via listen-and-learn, in: 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), 2023, pp. 236–241. doi:10.1109/CCNC51644.2023.10059624.

[14] T. D. Hoang, T. Oh, C. Park, I. Yun, Y. Kim, Llfuzz: an over-the-air dynamic testing framework for cellular baseband lower layers, in: Proceedings of the 34th USENIX Conference on Security Symposium, SEC '25, USENIX Association, USA, 2025.

[15] N. Silvanovich, How to hack shannon baseband (from a phone), in: OffensiveCon 2023, 2023. URL: https://www.youtube.com/watch?v=quw8SnmMWg4, talk presented at OffensiveCon; video available online.

[16] A. Cama, A walk with shannon, in: Infiltrate 2018, 2018. URL: https://infocon.org/cons/Infiltrate/Infiltrate%202018/Infiltrate%202018%20Slides/presentation.pdf, slide deck disponibile online at "Infiltrate 2018 Slides".

[17] W. Li, H. Wen, Z. Lin, Basemirror: Automatic reverse engineering of baseband commands from android's radio interface layer, in: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 2311–2325. URL: https://doi.org/10.1145/3658644.3690254. doi:10.1145/3658644.3690254.

[18] E. Kim, M. W. Baek, C. Park, D. Kim, Y. Kim, I. Yun, BASECOMP: A comparative analysis for integrity protection in cellular baseband software, in: 32nd USENIX Security Symposium (USENIX Security 23), USENIX Association, Anaheim, CA, 2023, pp. 3547–3563. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-eunsoo.

[19] E. Kim, D. Kim, C. Park, I. Yun, Y. Kim, Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols, 2021. doi:10.14722/ndss.2021.24365.

[20] W. Chieh Chao, C.-K. Chen, C.-M. Cheng, Cryfind: Using static analysis to identify cryptographic algorithms in binary executables, in: 2021 IEEE Conference on Dependable and Secure Computing (DSC), 2021, pp. 01–02. doi:10.1109/DSC49826.2021.9346229.

[21] polymorf, findcrypt-yara, https://github.com/polymorf/findcrypt-yara, 2022.

[22] L. Benedetti, A. Thierry, J. Francq, Detection of cryptographic algorithms with grap, Cryptology ePrint Archive, Paper 2017/1119, https://eprint.iacr.org/2017/1119, 2017.

[23] E. Firouzi, M. Ghafari, M. Ebrahimi, Chatgpt's potential in cryptography misuse detection: A comparative analysis with static analysis tools, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '24, ACM, 2024, p. 582–588. URL: http://dx.doi.org/10.1145/3674805.3695408. doi:10.1145/3674805.3695408.

[24] X. Shang, G. Chen, S. Cheng, S. Guo, Y. Zhang, W. Zhang, N. Yu, Foc: Figure out the cryptographic functions in stripped binaries with llms, 2025. URL: https://arxiv.org/abs/2403.18403. arXiv:2403.18403.

[25] G. Hernandez, M. Muench, T. Tucker, H. Searle, W. Zhu, P. Traynor, K. Butler, Emulating Samsung's Shannon Baseband for Security Testing, 2020.

[26] N. Group, Ghidra nanomips isa module, https://www.nccgroup.com/research-blog/ghidra-nanomips-isa-module/, 2024.

[27] Google, Android debug bridge (adb), https://developer.android.com/tools/adb, 2025. Android

Platform Tools.
[28] Hex-Rays, IDA Pro, https://hex-rays.com/ida-pro/, 2025. Accessed: 2025-12-05.
[29] National Security Agency, Ghidra Software Reverse Engineering Framework, https://ghidra-sre.org/, 2019. Accessed: 2025-12-05.
[30] Conviso, Reversing tips: (almost) automatically renaming functions with ghidra, https://blog.convisoappsec.com/en/automatically-renaming-functions-with-ghidra/, 2024.
[31] Hex-Rays, Ida f.l.i.r.t. technology description, https://docs.hex-rays.com/user-guide/signatures/flirt/ida-f.l.i.r.t.-technology-in-depth, 2024.
[32] National Security Agency, Ghidra BSIM description, https://ghidra.re/ghidra_docs/GhidraClass/BSim/BSimTutorial_Intro.html, 2019. Accessed: 2025-12-05.
[33] E. Barker, W. Burr, A. Jones, T. Polk, S. Rose, M. Smid, Q. Dang, et al., Recommendation for key management part 3: Application-specific key management guidance, NIST special publication 800 (2009) 57.